Universidad de Lleida
Escuela politécnica superior
Escuela técnica en informática de sistemas

Trabajo de fin de carrera

Procedimiento de diseño de circuitos digitales mediante FPGAs

Autor: Robert Antoni Buj Gelonch
Director: Francisco Clariá Sancho
Marzo 2007

Agradecimientos

Agradezco la colaboración directa o indirecta a todas las personas que de algún modo han contribuido en la elaboración del presente trabajo de final de carrera.

Mi más enérgico y sincero agradecimiento a mi director de trabajo de final de carrera Dr. Francisco Clariá Sancho, por sus acertados comentarios y pos su dedicación desmedida al trabajo de investigación que hemos compartido. A Jordi Palacín Roca por sus numerosas y desintersadas aportaciones.

Un agradecimiento especial a los miembros del departamento de diseño de hardware y software de la escuela superior de ciencias experimentales y tecnología de la universidad Rey Juan Carlos. Al Dr. José Ignacio Martínez Torres por haberme introducido al mundo del diseño de hardware con FPGAs. A los doctores Dr. Pablo Huerta Pellitero y Dr. Javier Castillo Villar, por haber compartido juntos tantos y buenos momentos.

También agradecer a Susie Cox por permitir la inclusión de la universidad politécnica superior de Lleida al CUP (Celoxica University Program), durante la realización de este proyecto de fin de carrera. Al equipo de Celoxica Ltd. de Oxford, especialmente a Johan Ditmar por las clases en el curso "Using Handel-C with DK" que realicé durante la realización del proyecto.

Al Dr. Álvaro Zamudio Lara del centro de investigaciones en ingeniería y ciencias aplicadas de la universidad del estado de Morelos (R.F. MEJICO), por la síntesis de algunos diseños y el acceso a herramientas de Xilinx.

Y por último, mi gratitud a Xavier Mas.

Índice de contenido

Índice de ilustraciones

Introducción

Durante las últimas décadas, el desarrollo de las nuevas tecnologías de fabricación de circuitos integrados, junto con el significativo avance de los equipos informáticos, ha proporcionado la aparición de nuevas arquitecturas electrónicas orientadas al prototipado de sistemas electrónicos, y a la aparición de sus herramientas destinadas a la síntesis y simulación.

El objetivo del presente trabajo de final de carrera es, presentar de forma general el procedimiento de diseño de circuitos con FPGAs, mediante el uso de herramientas de Celoxica Ltd.

Estructura del Trabajo de final de carrera

Este documento ha sido concebido con el propósito de ser una guía para estudiantes que cursen o hayan cursado los estudios en ingenierías técnicas de informática sistemas, electrónica o telecomunicaciones.

Al tratarse de una guía básica se detallan con precisión aspectos fundamentales reforzados con ejemplos y diagramas, omitiendo aspectos más complejos, llegando a no profundizar en ocasiones en determinados puntos. Ésta elección se tomó para no incurrir en el desbordamiento de información.

La estructura de este proyecto ha sido creada para no requerir conocimiento previo alguno sobre el diseño general de circuitos electrónicos, como tampoco acerca del diseño y programación de FPGAs. Los términos y conceptos necesarios serán presentados progresivamente a lo lago de éste documento.

Se ha decidido también incluir la librería estándar stdlib.hdl y los pines de la placa 203E como valor añadido, para que sirvan como base a una rápida consulta en la elaboración de nuevos diseños.

Organización

A continuación se presenta la organización por capítulos del trabajo de fin de carrera:

- Capítulo 1: Circuitos integrados lógicos
- Capítulo 2: Dispositivos lógicos programables
- Capítulo 3: FPGAs de Xilinx
- Capítulo 4: Placa de desarrollo RC203-E de Celoxica
- Capítulo 5: Diseño de sistemas electrónicos
- Capítulo 6: Diseño ESL de Celoxica
- Capítulo 7: Entorno de desarrollo: DK design suite
- Capítulo 8: Ejemplos

1 Circuitos integrados (CI) Lógicos

En primer lugar se van a ver con más detalle las memorias, posteriormente los PLD y por último las FPGA. Se hará un estudio introductorio a las memorias, como ejemplo de implementación de funciones lógicas en circuitos digitales. Concretamente se estudiarán las memorias PROM. A continuación se analizarán los PLDs comentando sus prestaciones y arquitectura , desde los más simples (SPLDs) hasta llegar a las FPGAs. La meta de dicho estudio es comprender la estructura y reconocer los componentes de las FPGAs.

De este modo se llegará a la conclusión que las FPGAs están compuestas principalmente de CPLDs, y a su vez, los CPLDs están compuestos de SPLDs. Además se observará que cada una de las arquitecturas anteriores (SPLD, CPLD, FPGA) pueden tener elementos de memoria específicos. Es por este motivo que las memorias serán el primer tipo de circuito digital objeto de estudio.

1.1 Clasificación

El siguiente esquema clasifica según su grado de programación los diferentes tipos de circuitos digitales.

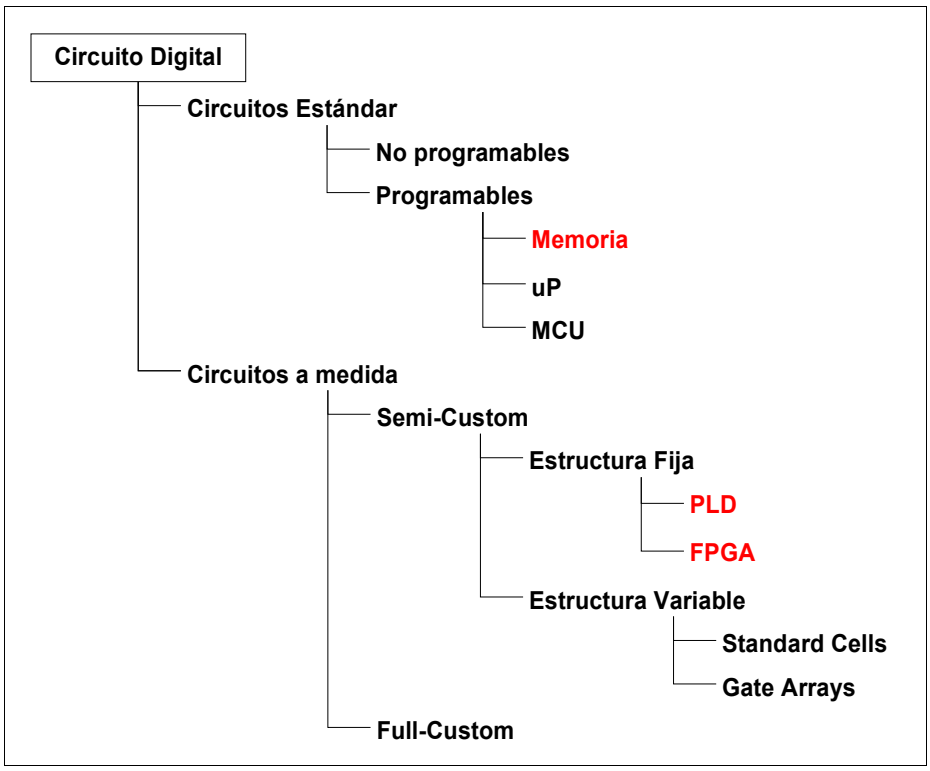


Ilustración 1: Circuitos digitales

1.2 Circuitos a medida

Un Circuito Integrado para Aplicaciones Específicas, o ASIC por sus siglas en inglés, es un circuito integrado hecho a la medida para un uso en particular, en vez de ser concebido para propósitos de uso general. Por ejemplo, un chip diseñado únicamente para ser usado en un teléfono celular es un ASIC. Cuando se realiza el diseño de un ASIC, en realidad se esta diseñando un sistema full-custom para ser producido en serie. El proceso de diseño de un ASIC termina cuando llega a la cadena de producción.

Por otro lado, los circuitos integrados de la serie 7400 son compuertas lógicas que se pueden utilizar para una multiplicidad de aplicaciones. En un lugar intermedio entre los ASIC y los productos de propósito general están los Productos Estándar para Aplicaciones Específicas, o ASSP por sus siglas en inglés.

Con los avances en la miniaturización y en las herramientas de diseño, la complejidad máxima, y por ende la funcionalidad, en un ASIC ha crecido desde 5.000 compuertas lógicas a más de 100 millones. Los ASIC modernos a menudo incluyen procesadores de 32-bit, bloques de memoria RAM, ROM, EEPROM y Flash, así como otros tipos de módulos. Este tipo de ASIC frecuentemente es llamado Sistema en un Chip, o SoC, por sus siglas en inglés. Los diseñadores de

ASIC digitales usan lenguajes descriptores de hardware (HDL), tales como Verilog o VHDL, para describir la funcionalidad de estos dispositivos.

Las matrices de compuertas programables, o FPGA por sus siglas en inglés, son la versión moderna de los prototipos con compuertas lógicas de la serie 7400. Contienen bloques lógicos e interconexiones, ambos programables, que permiten a un mismo modelo de FPGA, ser usada en muchas aplicaciones distintas.

Para los diseños más pequeños o con volúmenes de producción más bajos, las FPGAs pueden tener un coste menor que un diseño equivalente basado en ASIC, debido a que el coste fijo puede llegar a cientos de miles de euros, entendiendo por coste fijo el capital mínimo para preparar una línea de producción para la fabricación de un ASIC en particular.

C. DE APLICACIÓN ESPECÍFICA (ASIC)
CI.

- Ventajas:
  - silicio mejor aprovechado → menor espacio
  - menor número de CI → menor consumo
  - mayor rapidez → mayor frecuencia de funcionamiento
  - protección ante copia

- Campos de aplicación:
  - industria informática
  - industria de las telecomunicaciones
  - industria del automóvil
  - ... otros

CIRCUITOS SEMICUSTOM

- Siempre hay alguna parte del C.I. que el usuario no puede diseñar:
- Biblioteca de células estándar (Standard Cell Library). Se dispone de una biblioteca de células prediseñadas que el usuario elige, coloca e interconecta.
- Redes de Puertas (Gate Array). Matriz de puertas predefinida. El usuario interconecta las puertas.
- Dispositivos lógicos programables (PLD). El fabricante proporciona un circuito pre-procesado totalmente para que el usuario lo programe.

CIRCUITOS TOTALMENTE A MEDIDA (FULL CUSTOM)

- Se diseña todo.
- Ventajas:
  - gran densidad de integración → tamaño mínimo del dado
  - máxima fiabilidad y prestaciones
- Inconvenientes:
  - máxima cantidad de trabajo → máximo tiempo de desarrollo
  - máximo coste

1.3 Memorias

Las memorias son dispositivos que son capaces de proveer el medio físico para almacenar información.

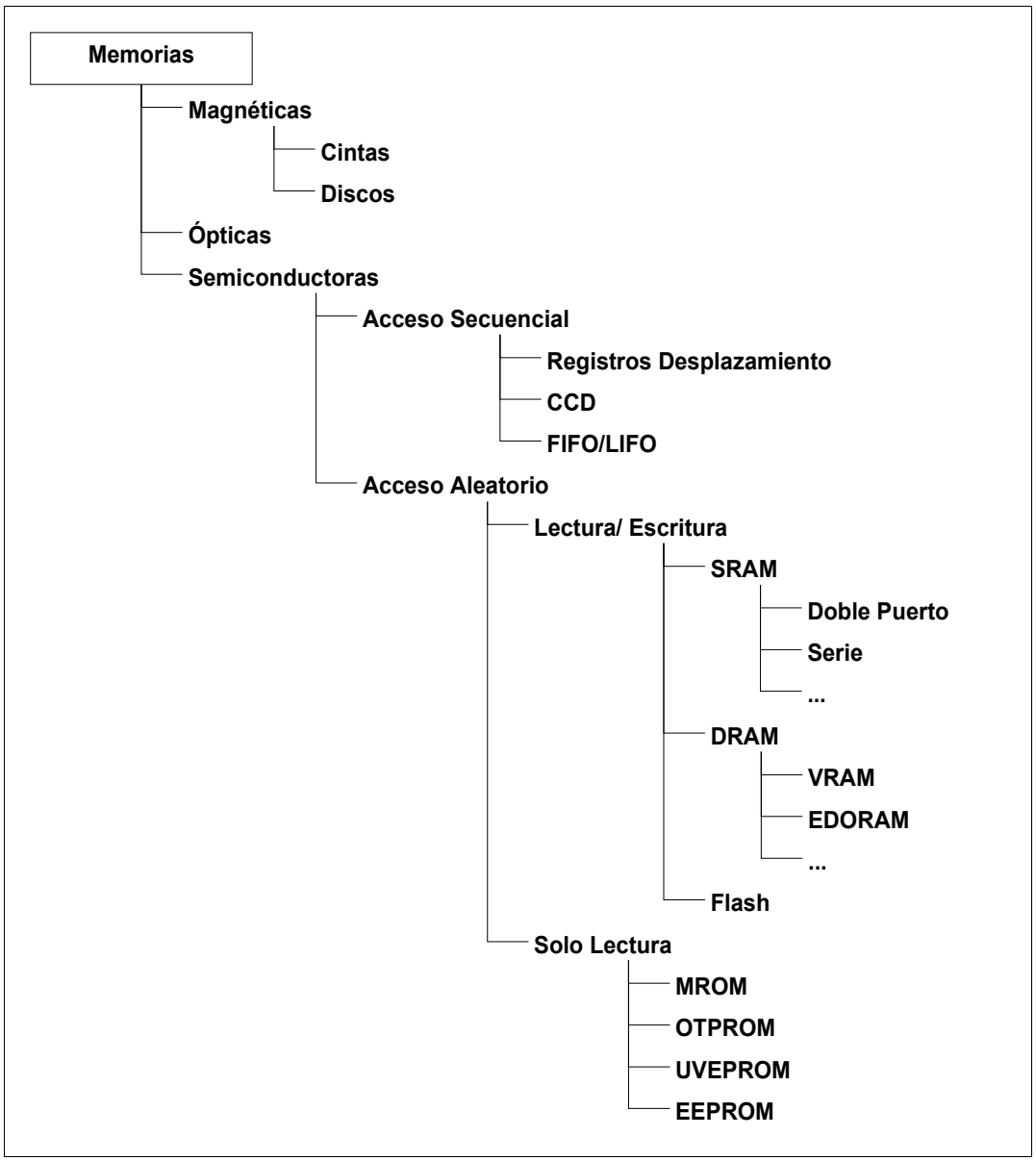


Ilustración 2: Tipos de memoria

Más del 90 % de las memorias se dedican a almacenar información. No obstante, también se pueden utilizar para la implementación de circuitos combinacionales, pueden sustituir la mayor parte de la lógica de un sistema. Las líneas de direcciones se utilizan como entrada y las de datos como salidas, implementan una tabla de verdad.

Las aplicaciones típicas de las memorias semiconductoras de acceso aleatorio son:

- Tablas
- Generadores de caracteres
- Convertidores de códigos
- Almacenamiento de datos

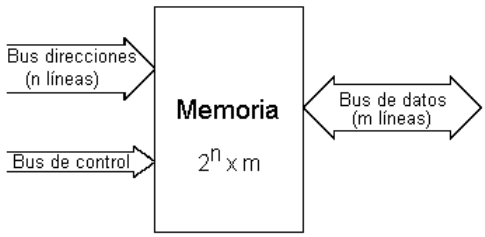
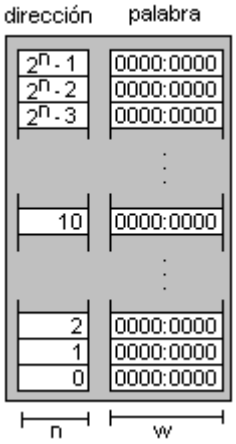


Ilustración 3: Buses de memoria

Mediante la dirección se puede acceder a la palabra almacenada en memoria. Con n líneas de dirección se pueden acceder a 2^n posiciones de memoria. El termino palabra hace referencia al número de bits almacenados en cada una de las posiciones de memoria. Existen diferentes anchos de palabra. Los más comunes son de 8, 10, 12, 16, 32 bits.



La capacidad o tamaño máximo de almacenamiento de una memoria se determina mediante la ecuación:

Tmáx = (Espacio de direcciones) x (Tamaño palabra)

Capacidad=Tmax = (2^n) x (m)

Donde,

n = número de líneas de direcciones
m = tamaño de la palabra

Organización: (2^n) “x” (m)

Ejemplo: Memoria con 12 líneas de direcciones y 8 de datos

Organización = 2^{12}x8= (2^2 x 2^{10}) x 8 = 4 K x 8 bits
Capacidad = 32.768 bits

Las celdas de memoria son las responsables del almacenamiento de datos en una memoria. El número de celdas es igual a la capacidad de la memoria. Cada celda de memoria almacena un bit.

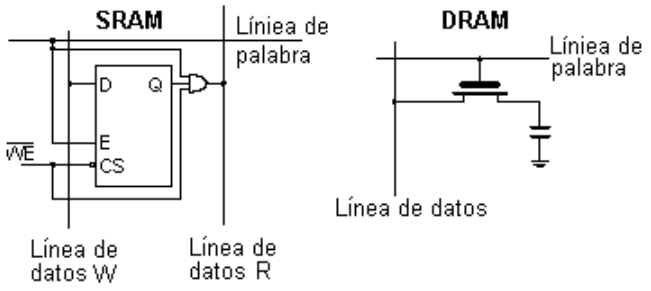


Ilustración 4: Celda de memoria SRAM y DRAM

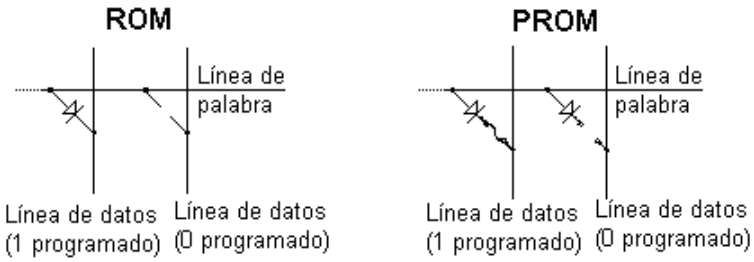
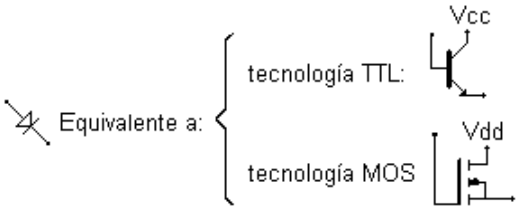


Ilustración 5: Celda de memoria ROM y PROM



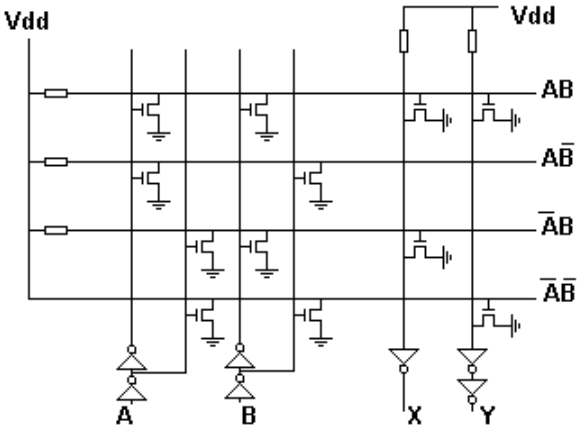
1.3.1 Read Only Memory, MROM y PROM

Las memorias de solo lectura, o ROM por sus siglas en inglés, al igual que las memorias de acceso aleatorio no volátil, o NVRAM por sus siglas en inglés, se caracterizan por no cambiar su información cuando se desconecta la fuente de energía.

Principalmente existen dos tipos de memoria ROM:

1. No programables por el usuario: **Mask ROM (MROM)**

Programadas por máscara en fábrica, proporcionan mejores prestaciones. Son las denominadas hardwired conectadas.



A	B	X	Y
1	1	0	1
1	0	1	0
0	1	0	0
0	0	1	1

Ilustración 6: MOS ROM

2. Programables por el usuario: **Programmable ROM (PROM)**

   También se conocen como **Field-programmable ROM** (**FPROM**)
   Proporcionan peores prestaciones, pero son menos costosas para volúmenes pequeños de producción y se pueden programar de manera inmediata.
   Según el número de veces en el que se pueda programar la memoria PROM:
   a. Programable solamente una vez, **One Time PROM (OTPROM)**. Generalmente son EVPROM con el cristal tapado.

   b. Programable n veces, **Erasable PROM (EPROM)**, según el método de borrado:
      i. Mediante luz ultra violeta, **Ultra Violet EPROM (UVEPROM)**

Ilustración 7: Memoria EV-PROM

      ii. Eléctricamente, **Electrical EPROM (EEPROM o E2PROM)**
         1. Direccionamiento serie
         2. Direccionamiento Paralelo

1.3.1.1 Ejemplo E2PROM: decodificador display 7-seg

Se utilizará una memoria EEROM para decodificar un número binario y mostrarlo por un display de 7 segmentos cátodo común. Cuando en una patilla del display de 7 segmentos hay 5v ("1" lógico) el segmento correspondiente se iluminará.

Las líneas de direcciones actuarán como entrada del número en binario. En la salida tendremos la palabra almacenada en esa posición de memoria.

Ejemplo de esquemático:

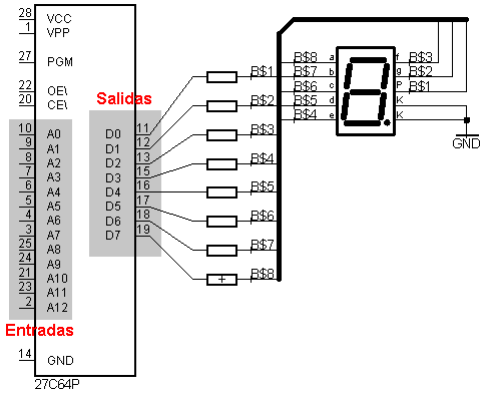


Ilustración 8: Decodificador Display 7 segmentos mediante E2PROM

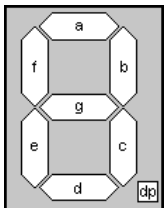
Display	Número (HEX)	a	b	c	d	e	F	g	dp
	0	1	1	1	1	1	1		
	1		1	1					
	2	1	1		1	1		1	
	3	1	1	1	1		1		
	4		1	1			1	1	
	5	1		1	1		1	1	
	6	1		1	1	1	1	1	
	7	1	1	1					
	8	1	1	1	1	1	1	1	
	9	1	1	1			1	1	
	A	1	1	1		1	1	1	
	B			1	1	1	1	1	
	C	1			1	1	1		
	D		1	1	1	1		1	
	E	1			1	1	1	1	
	F	1				1	1	1	
	Entrada	Salida							

FIG: Tabla de conversión

2 Dispositivos Lógicos Programables (PLD)

Este capítulo comienza con una clasificación de los dispositivos programables. Se verán cada una de las ventajas y desventajas proporcionadas para cada una de las arquitecturas de dicha clasificación, y a su vez aparecerán los elementos comunes o disyuntivos que existen entre las diversas arquitecturas.

Además aparecerán los elementos que se pueden programarse o no, e irán apareciendo esquemas que resumirán de un modo gráfico facilitando la comprensión. Al principio los esquemas están compuestos por puertas lógicas simples (AND, OR, NOT,..). A estos esquemas se les van añadiendo progresivamente más elementos, creciendo su funcionalidad.

La meta de este capítulo es llegar a las FPGA habiendo pasando por elementos más simples, facilitando de este como su comprensión, proporcionando además una visión general acerca de los dispositivos lógicos programables.

2.1 Clasificación

Un dispositivo lógico programable, o PLD por sus siglas en inglés, es un dispositivo cuyas características pueden ser modificadas y almacenadas mediante programación. Los PLDs se pueden clasificar en:

- PLDs combinatorios:
  - Constituidos por arreglos de compuertas *AND – OR*.
  - El usuario define las interconexiones y en esto consiste la programación.

- PLDs secuenciales:
  - Además de los arreglos de compuertas, incluyen *flip – flops* para programar funciones secuenciales como contadores y máquinas de estado.

Siendo los elementos lógicos constituyentes en los PLD:

- Secuenciales
  - AND
  - OR
  - NOT

- Combinacionales
  - FLIP-FLOP
  - LATCH

Los principales dispositivos que pertenecen a los PLDs son:

- ROMs
- PLAs
- PALs
- GALs
- CPLDs
- FPGAs
- ASICs

2.2 PLDs Simples (Simple PLD – SPLD)

Mediante la suma de productos se obtiene la función lógica a desarrollar. El circuito interno de un SPLD consiste en:

- Una matriz de conexiones
- Productos : arreglos de puertas **AND**
- Sumas: arreglos de puertas **OR**

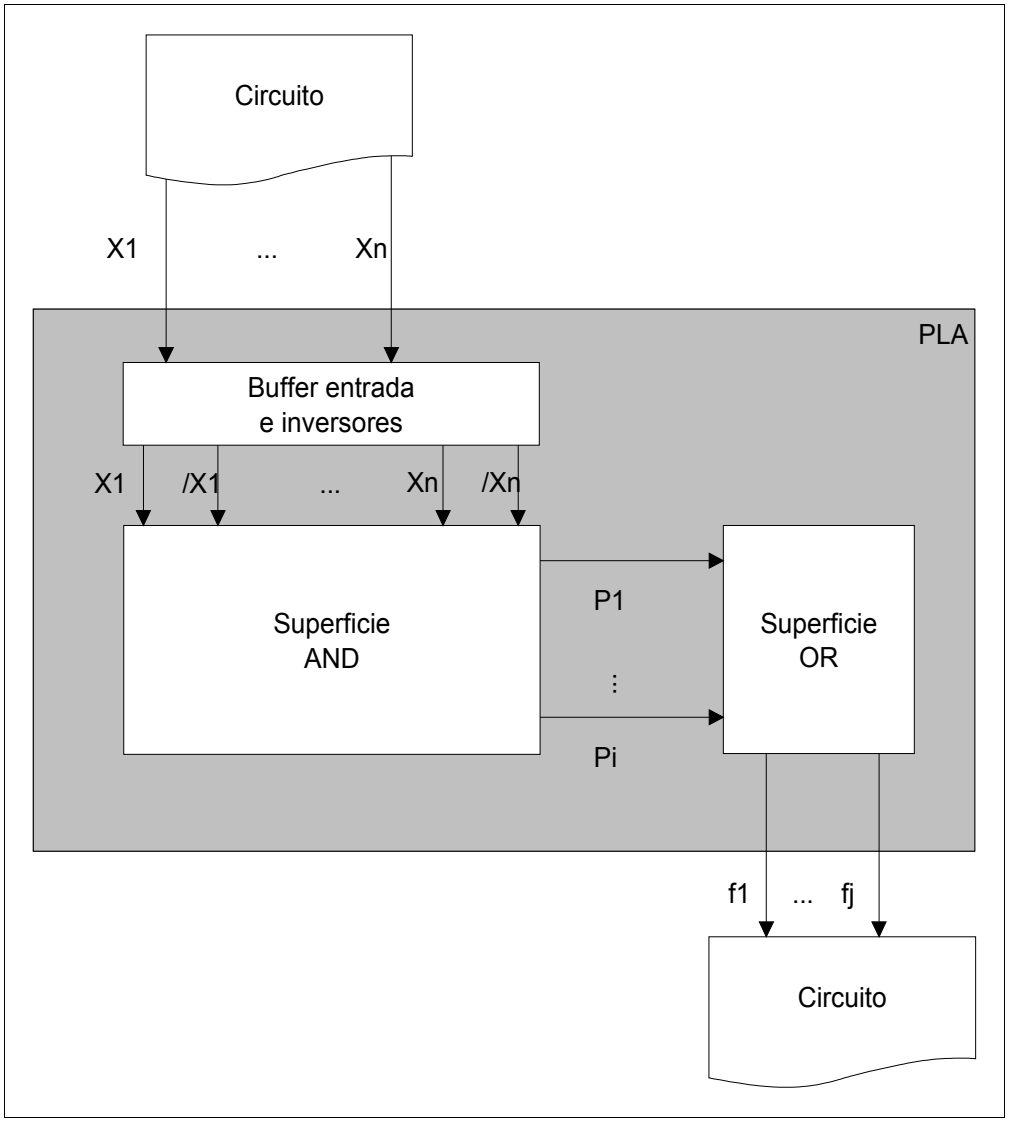


Ilustración 9: Diagrama genérico de un SPLD

2.2.1 Arquitectura e interconexiones programables

Una matriz de conexiones es una red de conductores distribuidos en filas y columnas, en los SPLDs, las interconexiones entre los conductores pueden ser fijas o programables. Los SPDL se pueden clasificar según el tipo de interconexiones:

Tipo SPLD				Puertas AND	Puertas OR	Programación	Tecnología
M			ROM	Fijas	Programables	Fábrica	Bipolar
OT		P				**1** sola vez	Bipolar
UV	E					**N** veces	MOS
E						**N** veces	MOS
PAL		PAL		Programables	Fijas	**1** vez	Bipolar
		PLS				**1** vez	Bipolar
		EPLD				**N** veces	MOS
		GAL				**N** veces	MOS
(F)PLA				Programables	Programables	**1 o N** veces	Bipolar/MOS

NOTA: EEPROM también se le conoce como E2PROM

PROM

- Puertas **AND** no programables
- Puertas **OR** programables
- Memoria

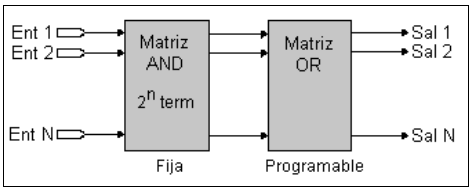


Ilustración 10: Estructura interna de una PROM

PLA

- Puertas **AND** programables
- Puertas **OR** programables

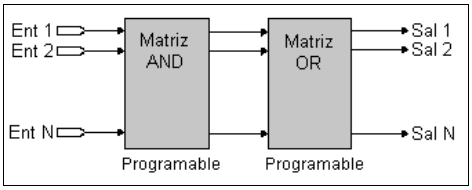


Ilustración 11: Estructura interna de una PLA

PAL

- Puertas **AND** programables
- Puertas **OR** no programables
- No reprogramable

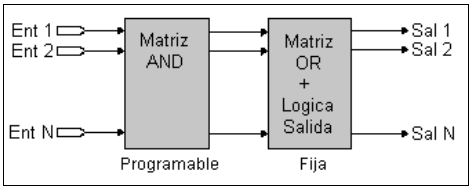


Ilustración 12: Estructura interna de una PAL

GAL

- Puertas **AND** programables
- Puertas **OR** no programables
- Reprogramable

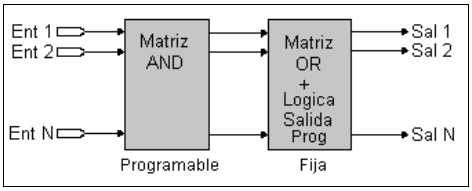
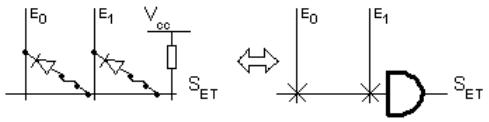


Ilustración 13: Estructura interna de una GAL

2.2.1.1 Matriz AND programable

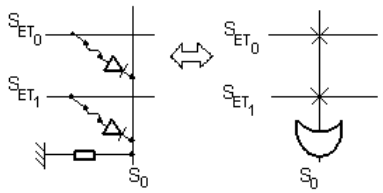


$SET = E_0 \text{ AND } E_1$

Ilustración 14: Matriz AND programable

- Se realiza la programación desconectando o no el "fusible".
- "E" representa el voltaje de entrada
- "SET" corresponde a la salida de la puerta lógica AND.

2.2.1.2 Matriz OR programable



$S_0 = SET_0 \text{ OR } SET_1$

Ilustración 15: Matriz OR programable

- Se realiza la programación desconectando o no el "fusible".
- "SET" representa el voltaje de entrada.
- "S" corresponde a la salida de la puerta lógica OR.

2.2.2 PLA o FPLA

El arreglo lógico programable, o PLA por sus siglas en inglés (Programmable Logic Array), es el dispositivo lógico programable más simple, a este dispositivo también se le suele llamar Field Programmable Logic Array , o FPLA por sus siglas en inglés.

Mediante un fusible se seleccionan las entradas del dispositivo que serán conectas a la puerta AND; las salidas de las puertas AND son conectadas a puertas OR también mediante un fusible, de esta manera se obtiene una función lógica en forma de suma de productos.

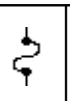
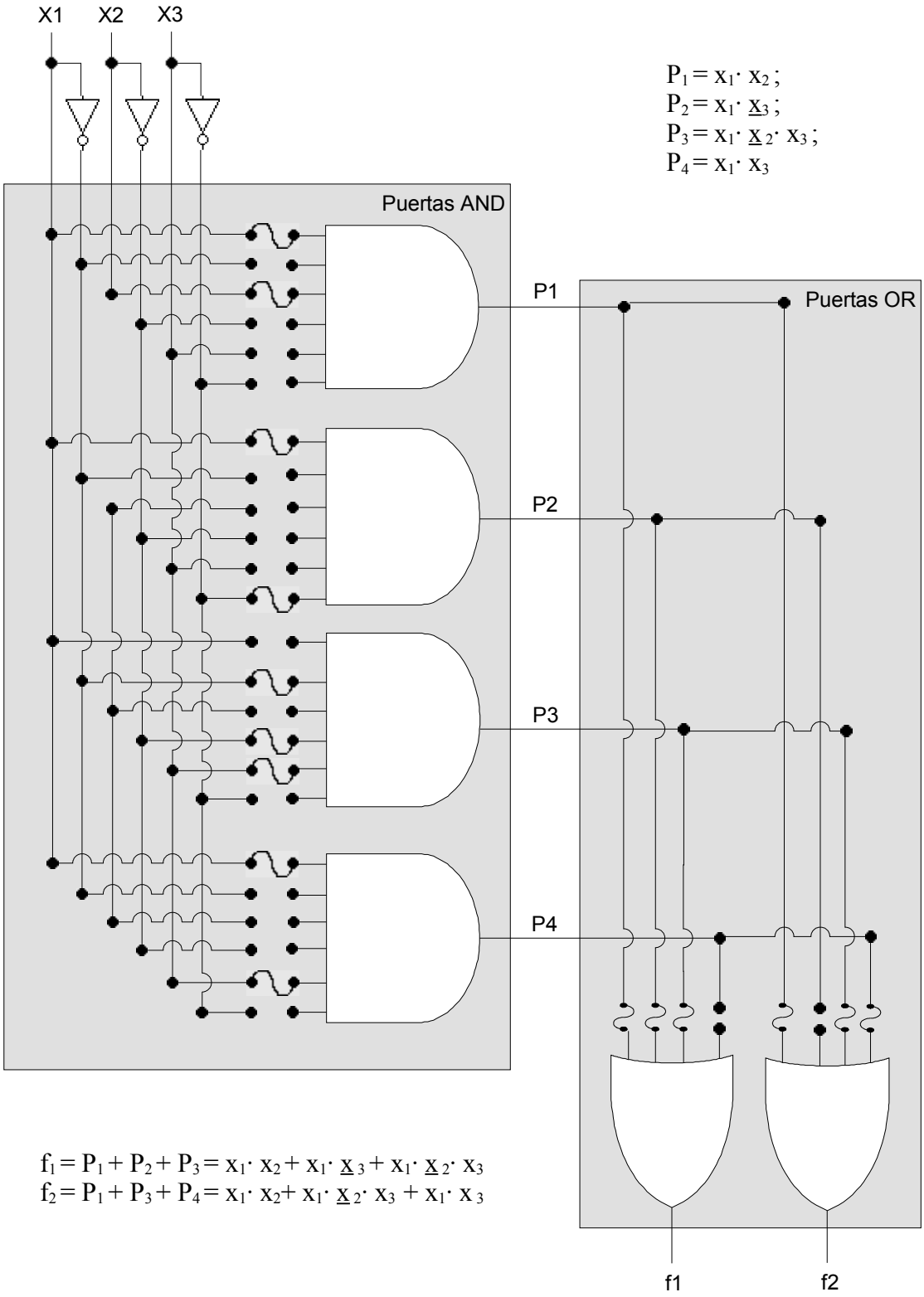
	Fusible cerrado
	Fusible abierto

Fig: Fusible

El siguiente esquema muestra el principio de funcionamiento de un PLA. El esquema muestra una PLA programado con unas determinadas funciones, **f1** y **f2**. La salida dichas funciones depende del valor de la entrada: **X1**, **X2** y **X3**. La salida

de una función se determina mediante la suma selectiva de productos específicos, **P1**, **P2**, **P3** y **P4**.



$P_1 = x_1 \cdot x_2$;
$P_2 = x_1 \cdot \underline{x}_3$;
$P_3 = x_1 \cdot \underline{x}_2 \cdot x_3$;
$P_4 = x_1 \cdot x_3$

$f_1 = P_1 + P_2 + P_3 = x_1 \cdot x_2 + x_1 \cdot \underline{x}_3 + x_1 \cdot \underline{x}_2 \cdot x_3$
$f_2 = P_1 + P_3 + P_4 = x_1 \cdot x_2 + x_1 \cdot \underline{x}_2 \cdot x_3 + x_1 \cdot x_3$

Debido a la complejidad y tamaño del esquema anterior para representar el esquema interno hardware programado de una PLA, este se suele simplificar. En la figura siguiente se muestra el esquema anterior simplificado, donde sólo se marcan las interconexiones programadas.

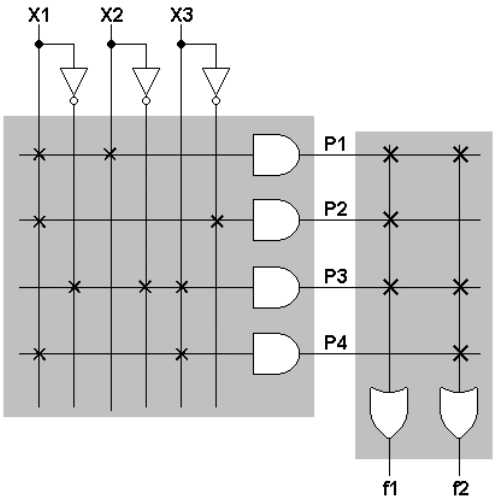


Ilustración 16: PLA 3 entradas 2 salidas

2.2.3 Matriz Lógica Programable (Programmable Array Logic)

En las **PLA** ambos arreglos (planos **AND y OR**) son programables. Esto tiene dos dificultades para el fabricante. En primer lugar es de fabricación compleja por la cantidad de llaves, y por otro lado requieren más tiempo. Estos problemas incentivaron a los fabricantes a realizar una mejora en el diseño, desarrollando un dispositivo de arreglos **AND** programables y **OR** fijos. Esta estructura se denomina Programmable Array Logic, la **PAL** ofrece menos flexibilidad que la **PLA** debido a que tiene un arreglo fijo (**OR**). En el ejemplo anterior la **PLA** permitía hasta 4 términos producto por compuerta **OR**, mientras que la **PAL** con el mismo número de productos, permite solo 2.

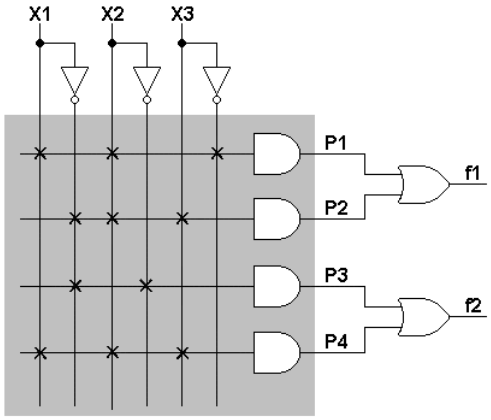


Ilustración 17: PAL 3 entradas 2 salidas

En el siguiente ejemplo se observa una PAL de 4 entradas con 4 salidas, donde cada puerta fija "OR" tiene cuatro entradas producto, puertas "AND" programables.

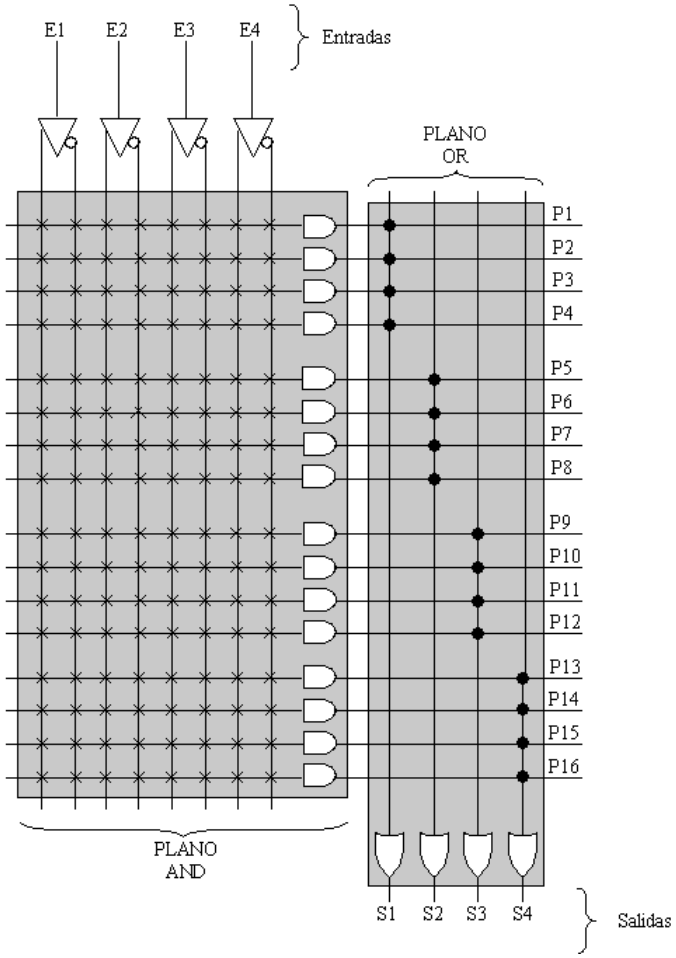


Ilustración 18: PAL 4 entradas 4 salidas

	Conexión programable
	Conexión fija
P	Producto
S	Salida
E	Entrada

2.2.3.1 Lógica de salida

La lógica de salida de los dispositivos PAL es fija. No tiene ningún plano "OR" programable, es por este motivo que existen varias configuraciones "OR" fijas.

El siguiente ejemplo muestra el esquema de un PAL simplificado, que implementa la función:

$$O = \hat{I}_2\hat{I}_1\hat{I}_0 + \hat{I}_2 I_1 + I_1\hat{I}_0$$

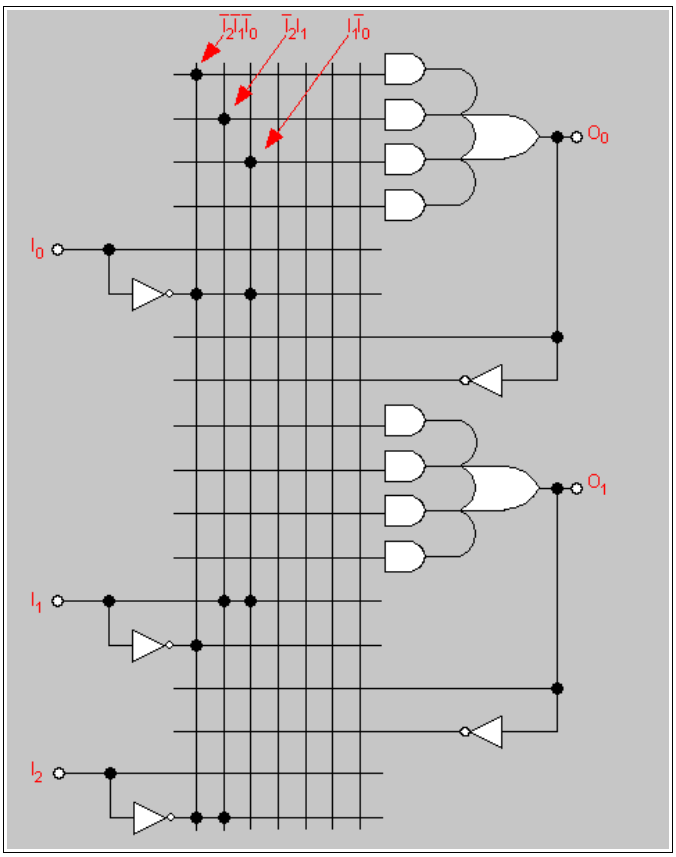


Ilustración 20: PAL

Donde cada columna representa un producto. Por ejemplo en la primera columna, $\hat{I}_2\hat{I}_1\hat{I}_0$ primero se selecciona el destino del producto (P_0 , P_1 , …) y a continuación se selecciona las entradas para ese producto, $\hat{I}_2$, $\hat{I}_1$, $\hat{I}_0$.

También se puede utilizar la salida de una puerta lógica como entrada. De este modo se pueden aumentar los términos de una función.

Por ejemplo se puede implementar la función:

$$O = I_2I_1I_0 + I_2I_1\hat{I}_0 + I_2\hat{I}_1I_0 + \hat{I}_2I_1I_0 + \hat{I}_2\hat{I}_1\hat{I}_0 + \hat{I}_2\hat{I}_1I_0$$

Mediante la división en dos operaciones:

$$O_1 = O_0 + \hat{I}_2\hat{I}_1\hat{I}_0 + \hat{I}_2\hat{I}_1I_0$$
$$O_0 = I_2I_1I_0 + I_2I_1\hat{I}_0 + I_2\hat{I}_1I_0 + \hat{I}_2I_1I_0$$

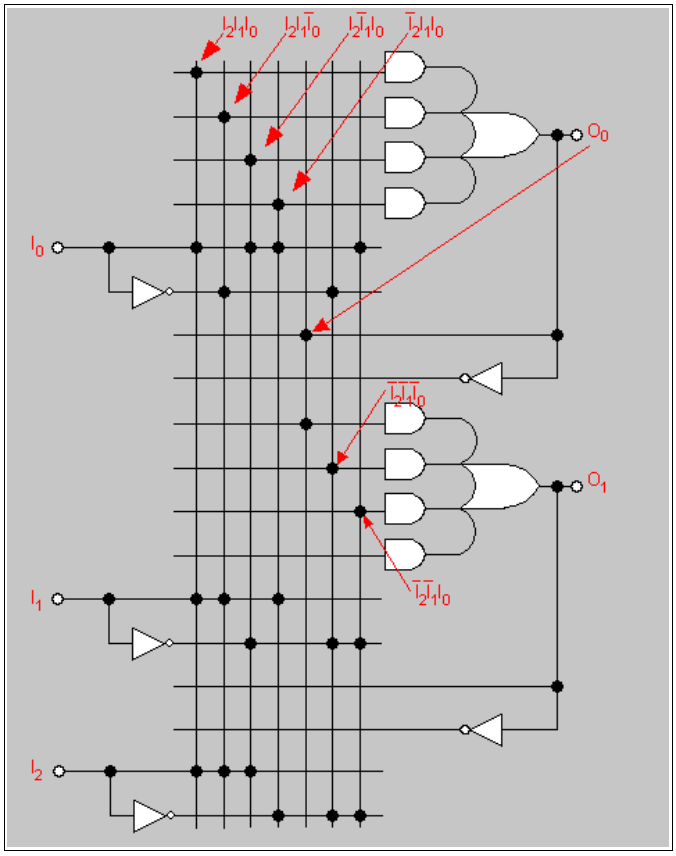


Ilustración 21: PAL con reentrada

De este modo aparece la lógica de salida de la puerta OR fija, y un conjunto de soluciones para reaprovechar la salida, ampliando las posibilidades para realizar funciones más complejas, es decir con más prestaciones.

La lógica de salida puede ser:

1. Combinacional
    a. Como salida combinacional
    b. Como entrada combinacional
    c. Como E/S combinacional
    d. Como E/S combinacional con polaridad
2. Secuencial
3. OLMC

2.2.3.1.1 Combinacional con control de salida

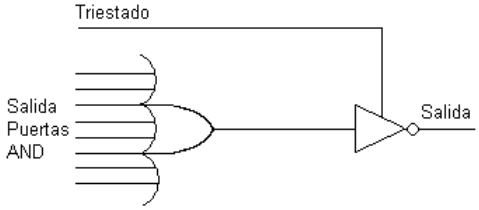


Ilustración 22: PAL, Configuración de salida combinacional

La más sencilla consiste en un arreglo de puertas combinacionales OR que están conectadas a la salida del plano AND, y el resultado de las puertas OR va directamente a la salida del PLD.

Una ampliación del esquema anterior, consiste en conectar la salida a la entrada.

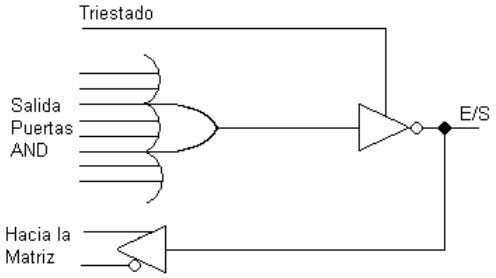


Ilustración 23: PAL, configuración como E/S

Una ampliación del esquema anterior, consiste en seleccionar la polaridad de salida, mediante una puerta XOR:

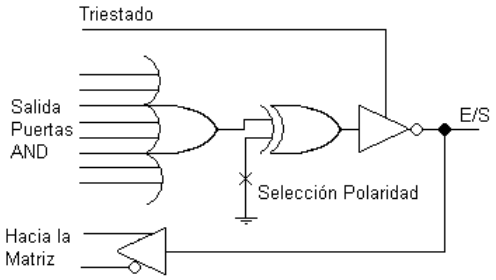


Ilustración 24: Pal, configuración de polaridad programable

2.2.3.1.2 Combinacional con control de salida mediante un producto

Consiste en asignar el producto de una función "AND" para realizar el control de la salida mediante un triestado. Una línea de la matriz "AND" va conectada al control del triestado.

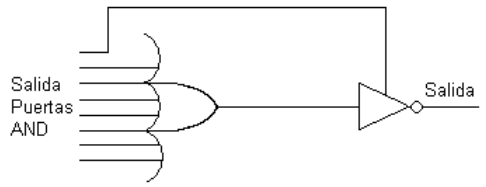


Ilustración 25: PAL, control tri-estado mediante un producto

2.2.3.1.3 Combinacional con salida siempre activa

Consiste en poner a encendido el triestado que controla la salida, de este modo siempre estará activa, sin ninguna entrada que la active o desactive.

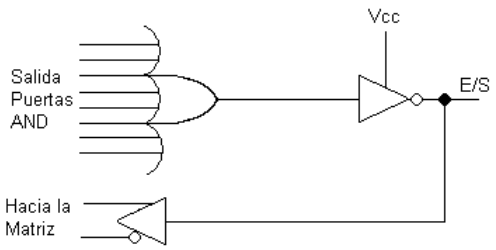


Ilustración 26: PAL, salida combinacional activa alto

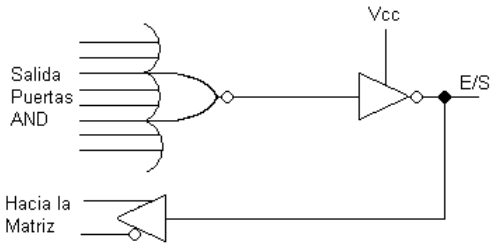


Ilustración 27: PAL, salida combinacional activa bajo

2.2.3.1.4 Combinacional como sólo entrada

La salida de la puerta OR está desactivada mediante el triestado.

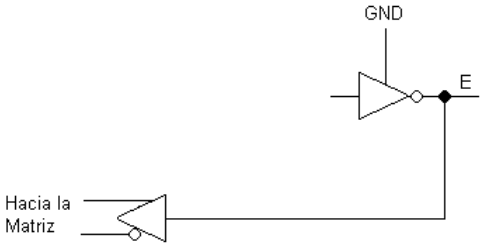


Ilustración 28: PAL, combinacional somo sólo entrada

2.2.3.1.5 Secuencial

Mediante el uso de un Flip-Flop de tipo D se almacena la salida en un registro.

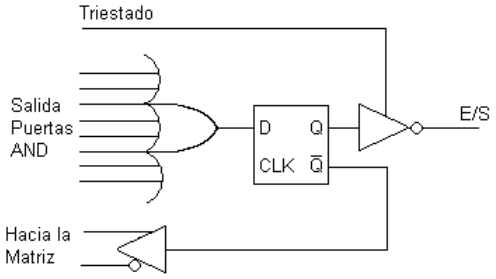


Ilustración 29: PAL, secuencial activo alto

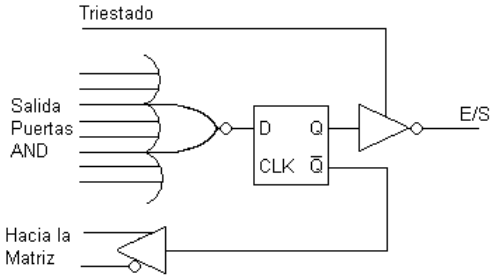


Ilustración 30: PAL, secuencial activo alto

2.2.3.1.6 OLMC (Output Logic Macro Cell)

Mediante el uso de un multiplexor se puede seleccionar, tanto la salida del circuito como la reentrada de su salida. Esto permite tener varias configuraciones posibles en una misma salida, reaprovechando el circuito. Para simplificar el circuito se ha eliminado la puerta XOR que permite seleccionar la polaridad, esta puerta iría detrás de la puerta OR, y su salida iría conectada: al multiplexor (10), al triestado, y como entrada del Flip-Flop (D).

La macro celda se utiliza en las PAL versátiles (llamadas V-PAL) y en las GAL. En las GAL la macro celda es reconfigurable, en las VPAL no.

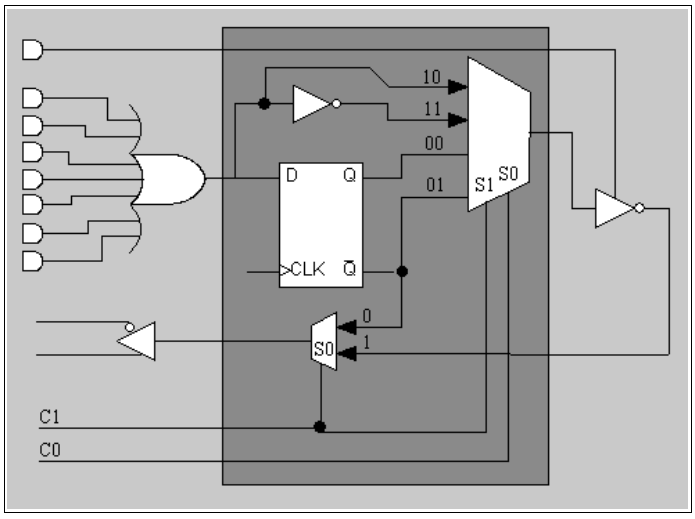


Ilustración 31: Macro celda

En una macro celda se puede configurar:

- La polaridad
- La función lógica
- La realimentación

Salida combinacional activa a nivel bajo (S1=1, S0=0):

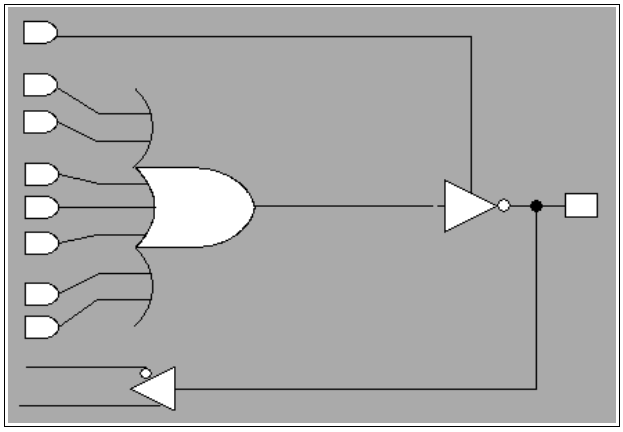


Ilustración 32: Macro celda, salida combinacional a nivel bajo

Salida combinacional activa a nivel alto (S1=1, S0=1):

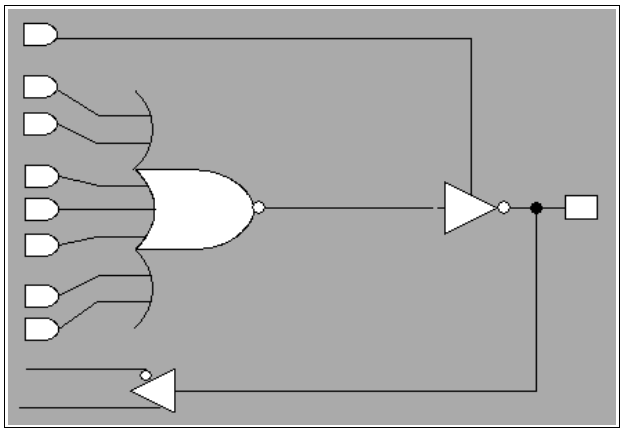


Ilustración 33: Macro celda, salida combinacional a nivel alto

Salida secuencial activa a nivel alto (S1=0, S0=1):

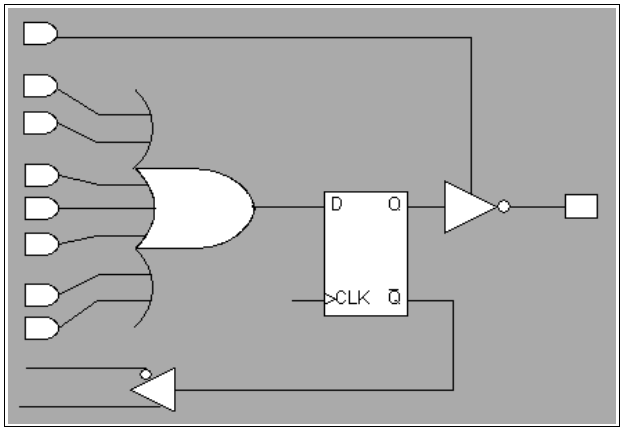


Ilustración 34: Macro celda, salida secuencial activa a nivel alto

Salida secuencial activa a nivel bajo (S1=0, S0=0):

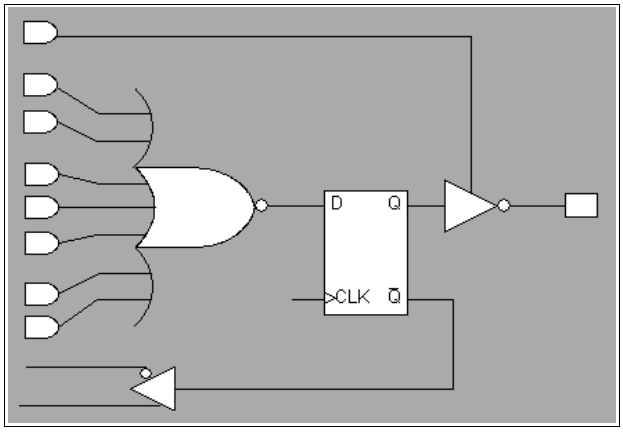


Ilustración 35: Macro celda, salda secuencial a nivel bajo

2.2.3.2 Ejemplo P16V8: decodificador display 7-seg CK

En este ejemplo se utiliza una PAL 16V8 para decodificar un número binario y mostrarlo por un display de 7 segmentos cátodo común. Cuando en una patilla del display de 7 segmentos hay 5v ("1" lógico) el segmento correspondiente se ilumina.

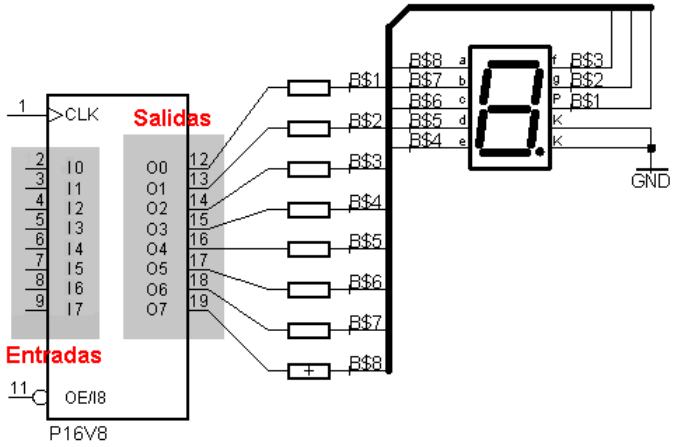
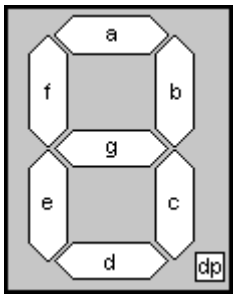


Ilustración 36: decodificador display7 segmentos P16V8

En la entrada tendremos el número en binario y en la salida del circuito combinacional del PAL tendremos el valor decodificado. Para implementar la función que decodificará el número binario para que salga en un display de 7 segmentos, hay varios métodos. Por su sencillez se ha escogido el lenguaje ABEL, sírvase como ejemplo, ya que este tipo de lenguajes posteriormente se verán con más detalle.

FIG: Tabla de conversión:

Display	Número (HEX)	a	b	c	d	e	F	g	dp
	0	1	1	1	1	1	1		
	1		1	1					
	2	1	1		1	1		1	
	3	1	1	1	1		1		
	4		1	1			1	1	
	5	1		1	1		1	1	
	6	1		1	1	1	1	1	
	7	1	1	1					
	8	1	1	1	1	1	1	1	
	9	1	1	1			1	1	
	A	1	1	1		1	1	1	
	B			1	1	1	1	1	
	C	1			1	1	1		
	D		1	1	1	1		1	
	E	1			1	1	1	1	
	F	1				1	1	1	
	Entrada	Salida							

```
module hex7seg
title 'Decodificador Hexadecimal – 7 Segmentos'
hex7seg device 'p16v6';
e3, e2, e1, e0 pin 2, 3, 4, 5;
a, b, c, d, e, f, g pin 19, 18, 17, 16, 15, 14, 13 istype 'com';
truth_table ([e3, e2, e1, e0] -> [a, b, c, d, e, f, g])
[0, 0, 0, 0] -> [1, 1, 1, 1, 1, 1, 0];
[0, 0, 0, 1] -> [0, 1, 1, 0, 0, 0, 0];
[0, 0, 1, 0] -> [1, 1, 0, 1, 1, 0, 1];
[0, 0, 1, 1] -> [1, 1, 1, 1, 0, 0, 1];
[0, 1, 0, 0] -> [0, 1, 1, 0, 0, 1, 1];
[0, 1, 0, 1] -> [1, 0, 1, 1, 0, 1, 1];
[0, 1, 1, 0] -> [1, 0, 1, 1, 1, 1, 1];
[0, 1, 1, 1] -> [1, 1, 1, 0, 0, 0, 0];
[1, 0, 0, 0] -> [1, 1, 1, 1, 1, 1, 1];
[1, 0, 0, 1] -> [1, 1, 1, 1, 0, 1, 1];
[1, 0, 1, 0] -> [1, 1, 1, 0, 1, 1, 1];
[1, 0, 1, 1] -> [0, 0, 1, 1, 1, 1, 1];
[1, 1, 0, 0] -> [0, 0, 1, 0, 1, 0, 1];
[1, 1, 0, 1] -> [0, 1, 1, 1, 1, 0, 1];
[1, 1, 1, 0] -> [1, 0, 0, 1, 1, 1, 1];
[1, 1, 1, 1] -> [1, 0, 0, 0, 1, 1, 1];
end hex7seg
```

FIG: Implementación (I) mediante tablas de verdad, lenguaje ABEL

```
module hex7seg
title 'Decodificador Hexadecimal – 7 Segmentos'
hex7seg device 'p16v6';
e3, e2, e1, e0 pin 2, 3, 4, 5;
a, b, c, d, e, f, g pin 19, 18, 17, 16, 15, 14, 13 istype 'com';
h, l = 1, 0;
equations
a = ( !e2&!e0 # e2&e1 # !e3&e2&e0 # !e3&e1&e0 # e3&!e2&!e1 );
b = ( !e2&!e0 #!e3&!e2 # !e3&!e1&!e0 # !e3&e1&e0 # e3&!e1&e0 );
c = ( !e1 # !e3&!e2 # e3&!e2 # !e3&e1 );
d = ( !e2&!e1&!e0 # e2&!e1&e0 # e3&!e2&e0 # !e3&!e2&e1 # e2&e1&!e0 );
e = ( !e2 & !e0 # e3 & e2 # e1 & !e0 # e3 & e1 );
f = ( e3&!e2 # e3&e1 # !e2&!e1&!e0 # !e3&e2&!e1 # e2&e1&!e0 );
g = ( e3 # e1&!e0 # e2&!e1 # !e2&e1 );
end hex7seg
```

FIG: Implementación (II) mediante ecuaciones, Lenguaje ABEL

2.2.4 Generic Array Logic

Para la implementación de la matriz AND programable se utiliza una memoria E2CMOS, por lo que se puede reprogramar varias veces, hecho que en las PAL no se puede realizar. La salida es reconfigurable mediante el uso de macro celdas como se ha comentado anteriormente en las V-PAL.

Generalmente para el diseño de circuitos con PAL se recurre a GAL en la fase de diseño/implementación, por la posibilidad de reutilización del mismo chip. Además las GAL pueden emular las PAL.

En la fase de producción se puede trasladar el diseño realizado en una GAL a una PAL. Además se puede proteger la propiedad intelectual del diseño rompiendo el fusible de programación haciendo imposible la lectura de su interior.

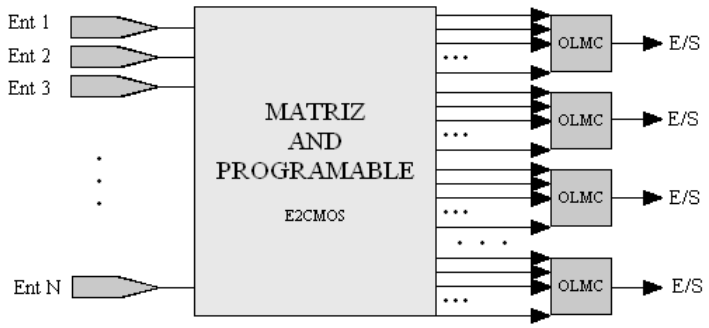


Ilustración 37: Diagrama de bloques internos en un GAL

2.2.5 Código de referencia estándar

Tipo	Tecnología	Nº entradas	Conf. Salida	Nº salidas
PAL	CE (1)	16	H	8

(1) campo tecnología

Erasable CMOS, programable n veces.
En blanco es tecnología bipolar, programable 1 sola vez

Configuraciones de salida más comunes:

- **H**: *salida a nivel lógico alto*
- **L**: *salida a nivel lógico bajo*
- **P**: *nivel lógico programable*
- **R**: *salida por registro*
- **RA**: *salida por registro asíncrono*
- **V**: *salida con producto de términos versátil*
- **VX**: *salida con producto de términos versátil con XOR*
- **X**: *salida por XOR con registro*
- **XP**: *salida por XOR con polaridad programable*

2.3 PLDs Complejos (Complex PLD - CPLD)

Los PLDs complejos contienen bloques lógicos (LBs) interconectados entre si por medio de un bloque programable (PIM). Los bloques lógicos (LBs) se interconectan con el exterior (I/O) por medio de bloques gestores de entrada/salida (IOBs).

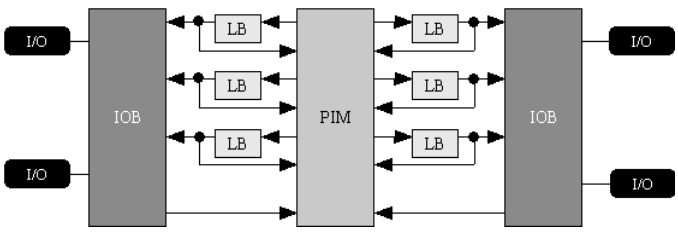


Ilustración 38: Diagrama de bloques internos en un CPLD

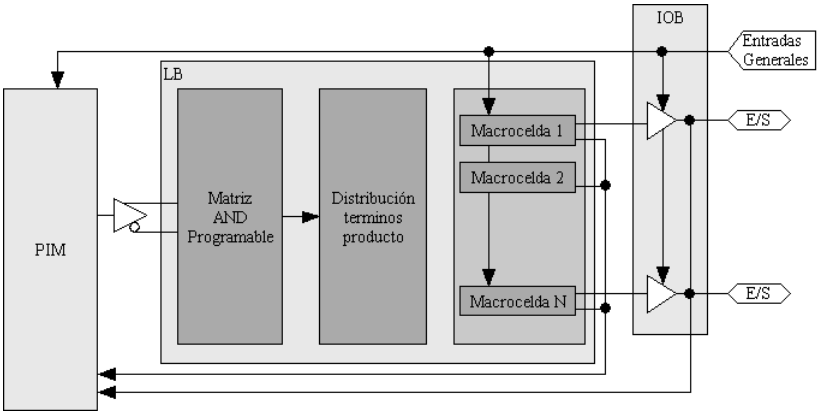


Ilustración 39: Diagrama de bloques internos PIM,LB,IOB de un CPLD

La capacidad de un CPLD depende de:

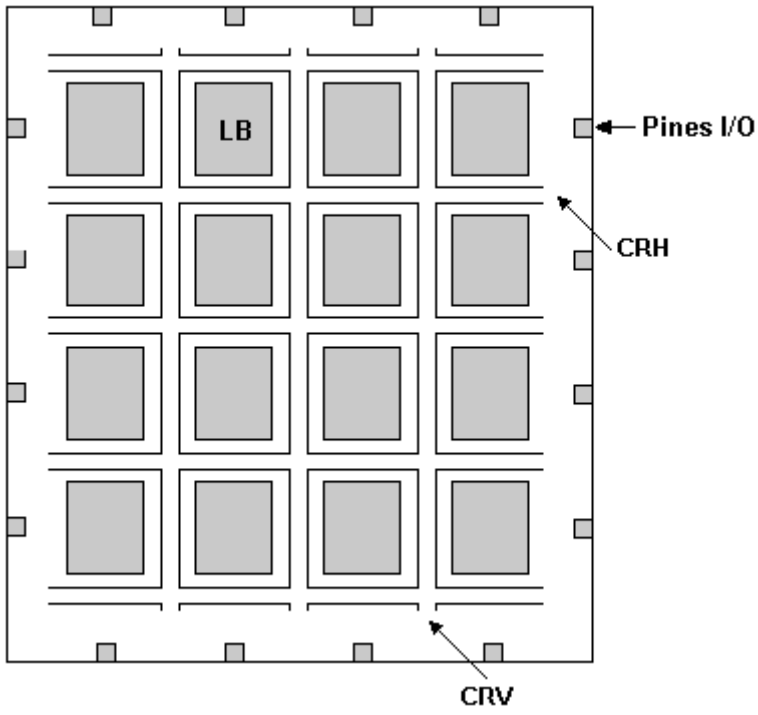
- Número de LB
- Número de entradas al LB
- Número de términos producto
- Número de macro celdas en cada LB
- Técnicas de distribución de términos producto
- Rutabilidad del PIM (probabilidad de una configuración que rutee las señales requeridas en los LB)

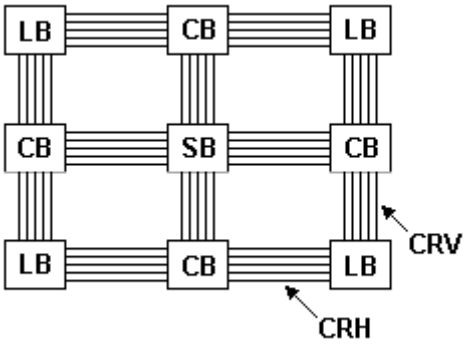
2.4 FPGA, Field Programmable Gate Array

FPGA (Field Programmable Gate Array: matrices de puertas programables por campo). - Estructura interna: matriz de bloques lógicos (LB) que se comunican entre sí y con los pines de E/S a través de canales de ruteo horizontal (CRH) y vertical (CRV).

Los diferentes modelos comerciales se diferencian en:

- Arquitectura de los LB
- Arquitectura de las interconexiones
- Tecnología de la programación





Los canales de ruteo están formados por pistas paralelas, que pueden estar segmentadas o no:

FIG: Interconexión en una FPGA con conexiones segmentadas (SB), (conexiones crosbar (Xbar, X), enrutadas (switch), …)

Los Bloque Lógicos (LB) implementan funciones lógicas, hay dos tipos:

1. Basados en multiplexores:

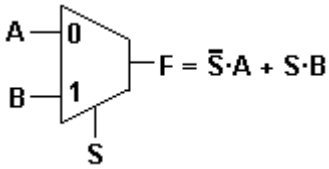


Ilustración 40: Lógica basada en multiplexor

2. Basados en LUT (look-ub tables):

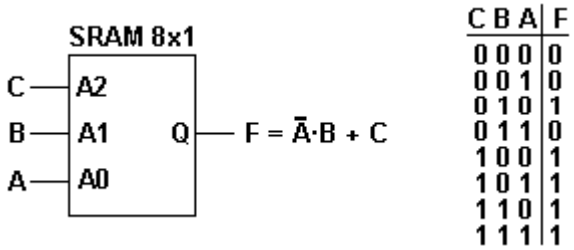


Ilustración 41: Lógica basada en LUT

El diseño de una aplicación con una FPGA se lleva a cabo especificando la función lógica a desarrollar, bien mediante un sistema CAD de dibujo de esquemas, bien mediante un lenguaje de descripción de hardware. Una vez definida la función a realizar, el diseño se traslada a la FPGA. Este proceso programa los bloques lógicos configurables (CLBs) para realizar una función específica (existen miles de bloques lógicos configurables en la FPGA). La configuración de estos bloques y la flexibilidad de sus interconexiones son las razones por las que se pueden conseguir diseños de gran complejidad. Las interconexiones permiten conectar los bloques lógicos (CLBs) entre sí. Finalmente, dispone de células de memoria de configuración (CMC, Configuration Memory Cell) distribuidas a lo largo de todo el chip, las cuales almacenan toda la información necesaria para programar los elementos programables mencionados anteriormente. Estas células de configuración suelen consistir en memoria RAM y son inicializadas en el proceso de carga del programa de configuración.

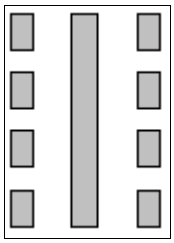
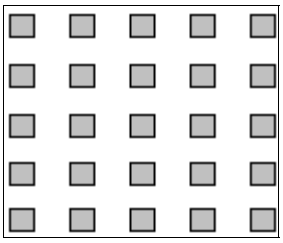
Los elementos programables de una FPGA son:

1. Bloques de lógica configurable (CLBs)
2. Bloques de entrada/salida (IOBs)
3. Interconexiones programables
   - Mediante tecnología fusible y ser del tipo OTP
   - Mediante antifusibles o mediante células tipo SRAM.

Tecnologías de programación de FPGA, tipos de CMC:

- Celdas SRAM (LCA): se pueden reprogramar pero son volátiles
- Celdas FLASH: se pueden reprogramar y no son volátiles
- Antifusibles: no son volátiles pero no se pueden reprogramar

2.4.1.1 FPGA vs. CPLD

Nombre	Complex Programmable Logic Devices	Field-Programmable Gate Array
Descripción	Circuitos Lógicos Prog. Complejos	Circuitos Prog. por el usuario en la aplicación
Organigrama		
Arquitectura	Tipo PAL	Tipo Matrices de puertas (GAL)
	Más Combinacional	Más Registros
Densidad	Baja-a-media	Medio-a-alta
Retardos	Predecibles	Dependiente de aplicación
Interconexión	Crossbar	Incremental

2.4.1.2 Estructura de una FPGA

Dependiendo del fabricante nos podemos encontrar con diferentes soluciones. Las FPGAs que existen actualmente en el mercado, dependiendo de la estructura que adoptan los bloques lógicos que tengan definidos, se pueden clasificar como pertenecientes a cuatro grandes familias.

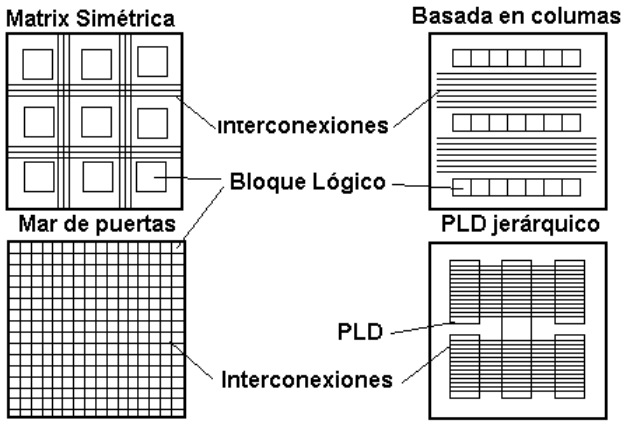


Ilustración 42: FPGA, Tipos de familias

- Matriz simétrica (symmetrical Array), XILINX
- Basada en columnas (Row-Based), ACTEL
- Mar de puertas (Sea-of-Gates), ORCA
- PLD jerárquico (Hierarchical PLD), ALTERA o CPLDs de XILINX.

Los elementos básicos de una FPGA son: (ejemplo FGPA de Xilinx)

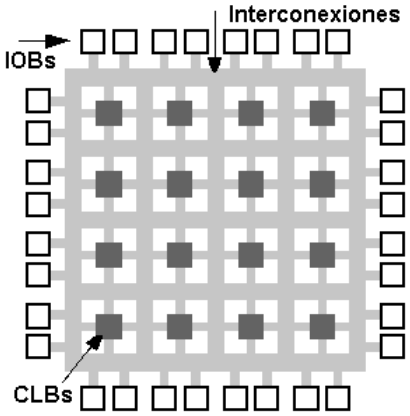


Ilustración 43: FPGA, elementos básicos

1. Bloques lógicos CLBs, cuya estructura y contenido se denomina arquitectura. Hay muchos tipos de arquitecturas, que varían principalmente en complejidad (desde una simple puerta hasta módulos más complejos o estructuras tipo SPLD). Suelen incluir biestables o MOS para facilitar la implementación de circuitos secuenciales. Otros módulos de importancia son los bloques de Entrada/Salida (IOB).

2. Recursos de interconexión, cuya estructura y contenido se denomina arquitectura de rutado.

3. Memoria RAM, que se carga durante el RESET para configurar bloques y conectarlos.

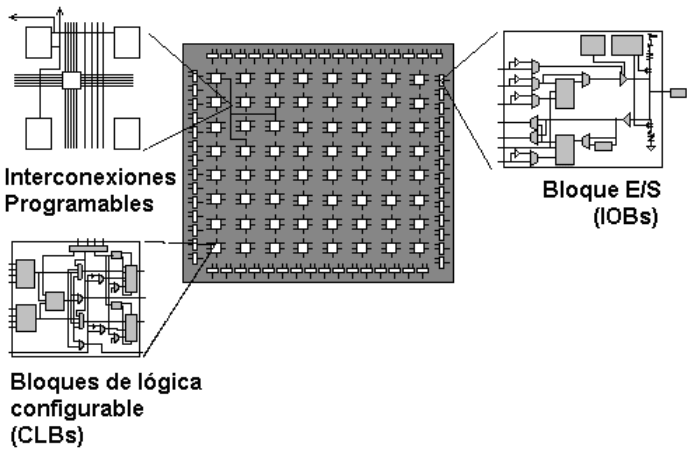


Ilustración 44: FPGA, elementos básicos en las FPGAs de Xilinx (Spartam/XC4000)

2.4.1.3 CLBs: Bloques Lógícos Configrables

Constituyen el núcleo de una FPGA.

Cada CLB presenta una sección de lógica combinacional programable y registros de almacenamiento:

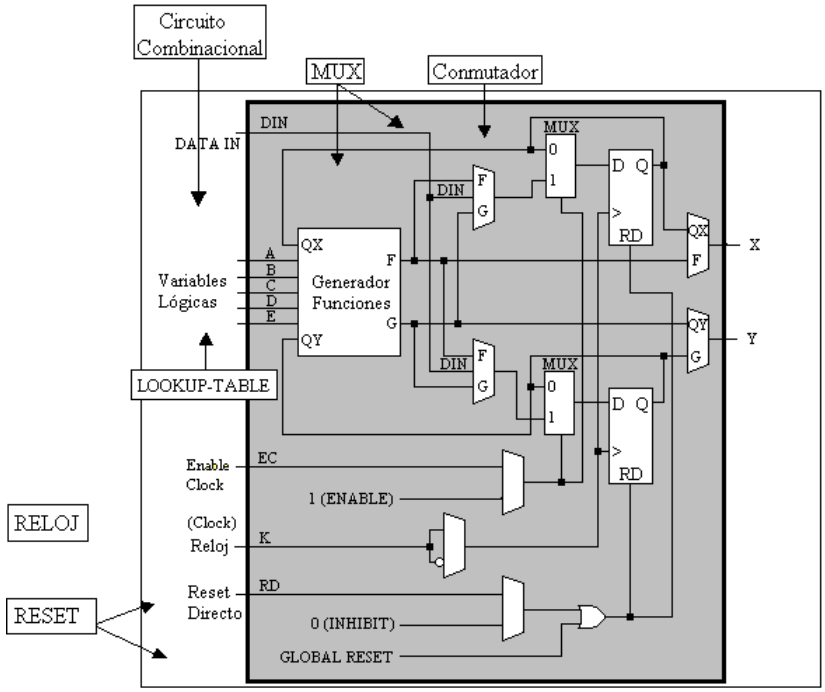


Ilustración 45: FPGA, bloques lógicos configurables (CLBs)

Los registros de almacenamiento sirven como herramientas en la creación de lógica secuencial.

La sección de lógica combinacional suele consistir en una LUT (Look Up Table), que permite implementar cualquier función bolean a partir de sus variables de entrada. Su contenido se define mediante las células de memoria (CMC).

Se presentan también multiplexores (MUX) y conmutadores, como elementos adicionales de direccionamiento de los datos del CLB, los cuales permiten variar el tipo de salidas (combinacionales o registradas), facilitan caminos de realimentación, o permiten cambiar las entradas de los biestables. Se encuentran controlados también por el contenido de las células de memoria (CMC).

2.4.1.4 IOBs: Bloques entrada/Salida

La periferia de la FPGA está constituida por bloques de entrada/salida configurables por el usuario.

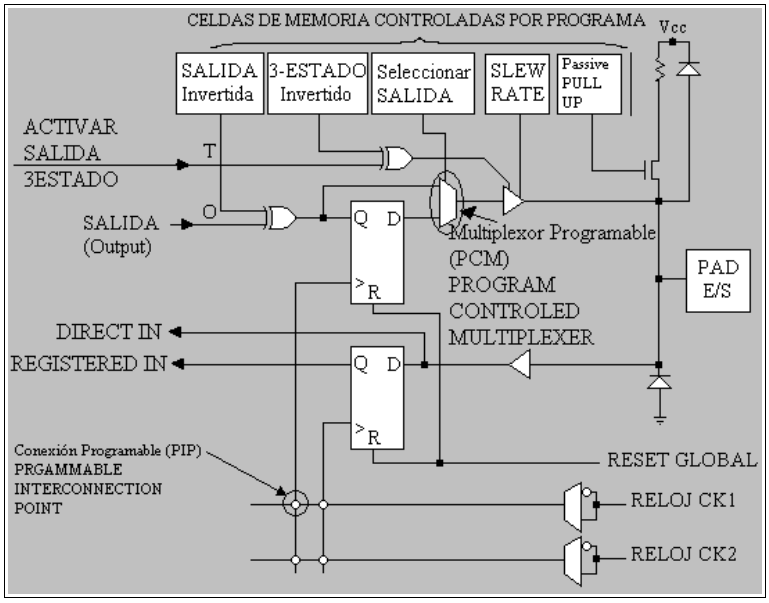


Ilustración 46: FPGA, bloques de entrada/salida (IOBs)

- Cada bloque puede ser configurado independientemente para funcionar como entrada, salida o bidireccional, admitiendo también la posibilidad de control triestado.
- Los IOBs pueden configurarse para trabajar con diferentes niveles lógicos (TTL, CMOS, …).
- Además, cada IOB incluye flip-flops que pueden utilizarse para registrar tanto las entradas como las salidas.

2.4.1.5 Lineas de interconexión

- Constituyen un conjunto de caminos que permiten conectar las entradas y salidas de los diferentes bloques.
- Están constituidas por líneas metálicas de dos capas que recorren horizontal y verticalmente las filas y columnas existentes entre los CLBs.

Dos elementos adicionales participan activamente en el proceso de conexión:

- Puntos de Interconexión Programable (PIP, Programmable Interconnection Point)
  - Permiten la conexión de CLBs e IOBs a líneas metálicas cercanas.
  - Consisten en transistores de paso controlados por un bit de configuración

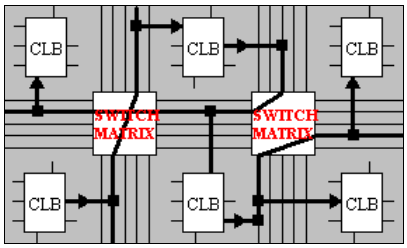
- Matrices de interconexión (SW, Switch Matriz o Magic Box).
  - Son dispositivos de conmutación distribuidos de forma uniforme por la FPGA.
  - Formados internamente de transistores que permiten la unión de las denominadas líneas de propósito general, permitiendo conectar señales de unas líneas a otras.

Tipo de líneas de interconexión:

1. Líneas de propósito general
2. Líneas directas
3. Líneas largas

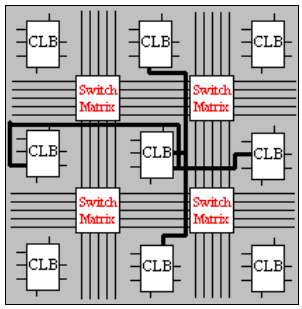
2.4.1.5.1 Líneas de propósito general

Conjunto de líneas horizontales y verticales conectadas a una matriz de interconexión.



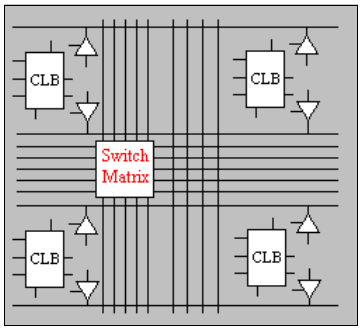
2.4.1.5.2 Líneas directas

Líneas de conexión directa entre bloques, sin tener que pasar por ninguna matriz de interconexión. Proporcionan la implementación más eficiente de redes entre CLBs e IOBs adyacentes, al introducir retardos mínimos y no usar recursos generales (SM).



2.4.1.5.3 Líneas largas

Líneas conductoras de señal de gran longitud que atraviesan la FPGA de arriba abajo y de izquierda a derecha. Permiten un fan-out elevado y un acceso rápido de una señal a un punto distante, sin la degradación temporal que originan las conexiones de propósito general. Suelen estar limitadas, por lo que es conveniente dedicarlas para señales críticas, como pueda ser el reloj o señales de inicialización globales.



2.4.1.6 Células de memoria de configuración (CMC)

- El dato de una célula de memoria habilita o deshabilita dentro del dispositivo un elemento programable.
- Están constituidas por dos inversores CMOS conectados para implementar un latch, además de un transistor de paso usado para leer y escribir la célula.
- Sólo son escritas durante la configuración y pueden ser leídas durante un proceso conocido como "readback".
- El proceso de configuración consiste en la carga de un fichero formado por un conjunto de bits, cada uno de los cuales es almacenado en una de estas células de memoria estática.

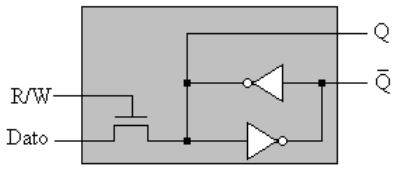


Ilustración 47: FPGA, CMC

3 FPGAs de Xilinx

Como se ha visto en el capítulo anterior, existen varios tipos de FPGAs. Ahora se analizará con más detalle un conjunto de FPGAs de Xilinx, concretamente a las familias pertenecientes a Virtex, de Xilinx. Se ha elegido este grupo de FPGAs de Xilinx ya que la placa de desarrollo sobre la cual se han realizado la mayoría de ejemplos utiliza una FPGA de Virtex-II, por este motivo no se detallan las arquitecturas posteriores. No obstante, serán de fácil comprensión ya que las arquitecturas más nuevas son una progresión de las aquí descritas.

Para cada una de las FPGAs de Virtex Xilinx que están en este capítulo se describen sus características principales, el tipo de tecnología y los bloques que incorpora, todo ello da paso a la denominación de arquitectura, y de aquí, que se detallen para comprender la funcionalidad de cada bloque y del número de elementos requeridos.

3.1 Virtex 2.5V

3.1.1 Principales características

- Densidad de puertas desde 50k hasta 1 Millón
- Soporta 16 intefaces estándar de alto rendimiento
- Rendimiento de sistema alrededor de 200MHz incluyendo E/S
- Compatible con PCI a 66MHz
- Circuitería para gestión de relojes:
  - 4 DLLs
  - 4 líneas principales para distribución de reloj
  - 24 líneas locales secundarias
- Sistema jerárquico de memoria:
  - LUTs
  - Block RAM
- Lógica especial para:
  - acarreo
  - multiplicadores
  - encadenamiento de funciones
- Arquitectura flexible:
  - acarreamiento rápido encadenado
  - ayuda del multiplicador
  - encadenamieno en cascada a la entrada de funciones
  - registros y latches abundantes
  - interconexiones internas con soporte de tri-estados
  - IEEE 1149.1
- Configuración del sistema basada en SRAM
  - reconfiguración parcial
- Proceso metálico de 5 capas a 0.22µm

3.1.2 Dispositivos pertenecientes

Device	CLB Array	Slices	Max I/O	BlockRAM 4kb	Distributed RAM bits
XCV50	16x24	768	180	8	24.576
XCV100	20x30	1.200	180	10	38.400
XCV150	24x36	1.768	260	12	55.296
XCV200	28x42	2.352	284	14	75.264
XCV300	32x48	3.072	316	16	98.304
XCV400	40x60	4.800	404	20	153.600
XCV600	48x72	6.912	512	24	221.184
XCV800	56x84	9.408	512	28	301.056
XCV1000	64x96	12.288	512	32	393.116

3.1.3 Arquitectura

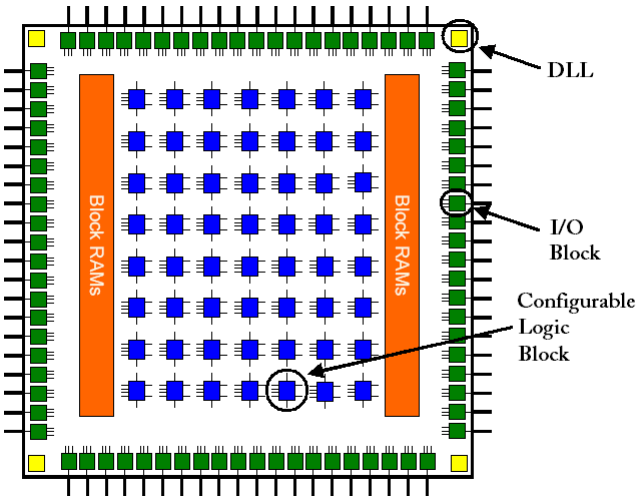


Ilustración 48: Arquitectura Virtex

- ***CLBs***, proporcionan los bloques funcionales para constituir la lógica
- ***IOBs***, proporciona el interfaz entre los pines y los CLBs
- ***DLLs***, compensación en la propagación del reloj

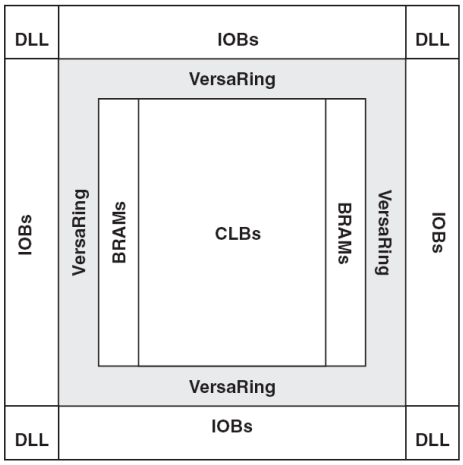


Ilustración 49: Virtex matriz de puertas programable por el usuario

- ***GRM*****, l**os CLBs se conectan a través de una matriz de enrutamiento general, o GRM de sus siglas en iglés (General routing matrix).
- ***VersaBlock*****, c**ada CLB se conecta a un VersaBlock que proporciona además interconexionado local para conectar el CLB a la GRM.
- ***VersaRing*****, e**l interfaz de E/S VersaRing proporciona interconexionado adicional alrededor del dispositivo.
- ***BlockRams*****, c**olumnas de bloques de memoria de 4096 bit/bloque.

3.1.4 2-Slice Virtex CLB

- Cada CLB tiene 2 Slices.
- Cada Slice tiene 2 LC (Logic Cell)
- Existe lógica adicional que permite contabilizar 1 CLB=4.5 LC

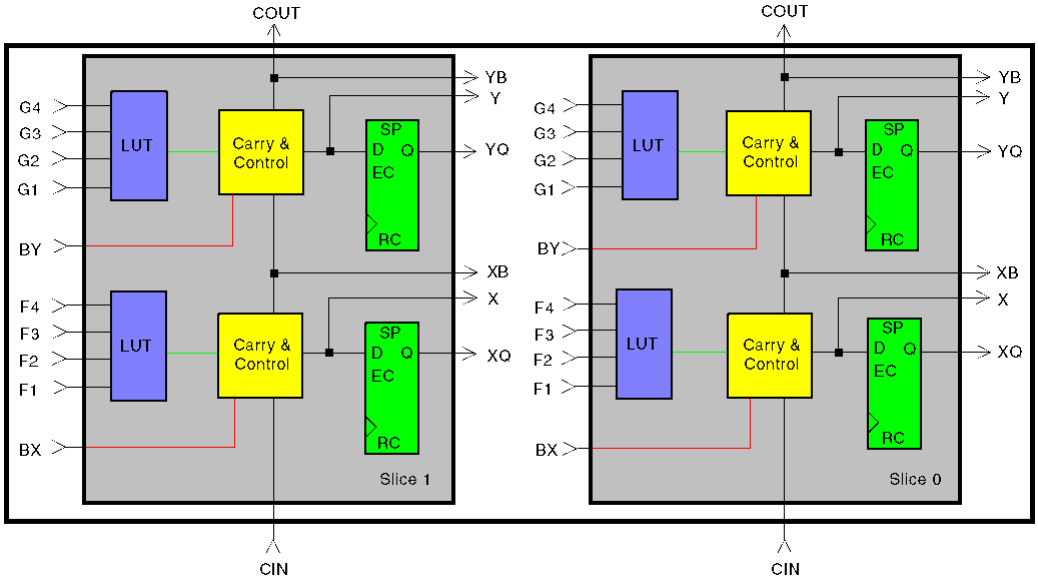


Ilustración 50: 2-Slice Virtex CLB

- Existen 2 señales programables de set/reset síncronas o asíncronas:
  - SR
  - BY
- Las señales de control se comparten por los dos flip-flop de un slice:
  - SR
  - CLK
  - CE
  - BY
- F5 combina las salidas de las dos LUTs de un slice:
  - LUT G
  - LUT H
- F6 combina las salidas de las 4 LUTs de un CLB.
  - F5
  - BY
  - FSIN
- 2 líneas de carry por CLB:
  - COUT
  - CIN
- XOR y AND dedicadas.
- 2 BUFT /CLB (Flip-Flop D).

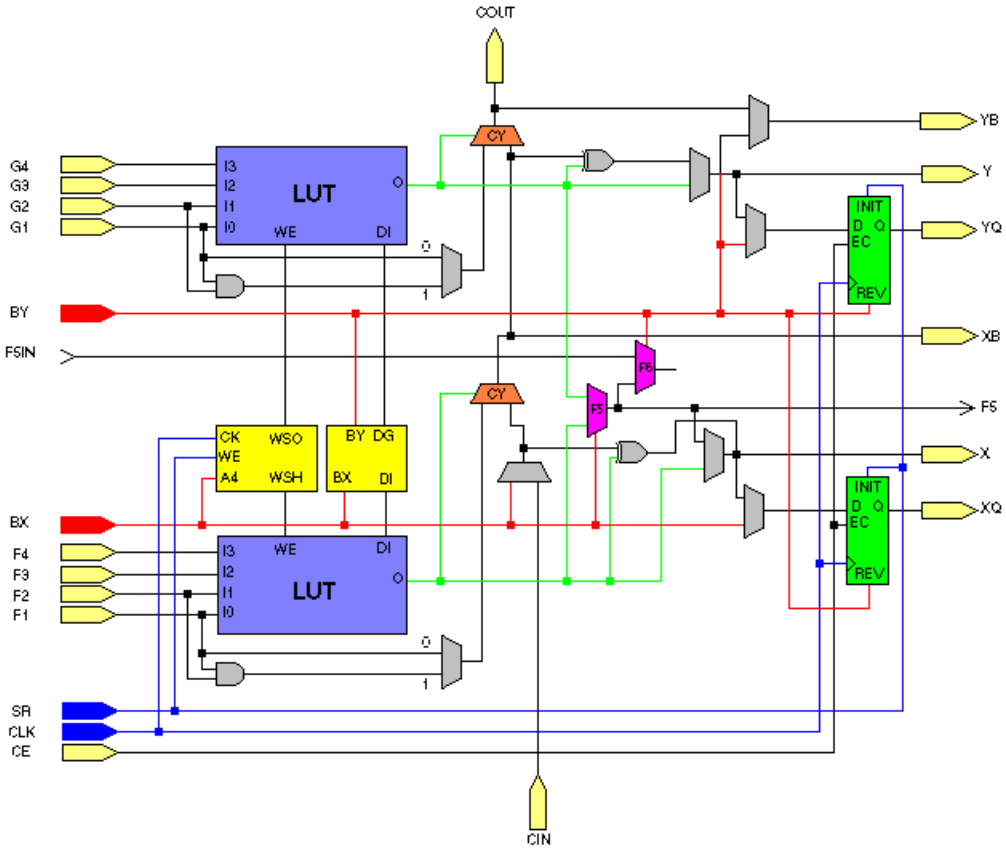


Ilustración 51: Slice Virtex CLB

3.2 Virtex-E 1.8V

3.2.1 Principales características

- Densidad de puertas desde 58k hasta 4 Millones.
- Rendimiento de sistema alrededor de 240MHz incluyendo E/S – un 30% más rápido que Virtex.
- Compatible con PCI 66MHz
- Soporta 20 intefaces estándar de alto rendimiento
- 8 DLLs – Cada uno puede multiplicar el reloj x4
- Sistema de memoria jerárquico
- Arquitectura flexible
    - acarreamiento rápido encadenado
    - ayuda del multiplicador
    - encadenamieno en cascada a la entrada de funciones
    - registros y latches abundantes
    - interconexiones internas con soporte de tri-estados
    - IEEE 1149.1
- Configuración del sistema basada en SRAM
    - reconfiguración parcial
- Proceso metálico de 6 capas a 0.18µm

3.2.2 Dispositivos pertenecientes

Dispositivo	CLB Array	Slices	Max I/O	BlockRAM 4kb	Distributed RAM bits
XCV50E	**16x24**	**768**	176	16	**24.576**
XCV100E	**20x30**	**1.200**	196	20	**38.400**
XCV200E	**28x42**	**2.352**	**284**	28	**75.264**
XCV300E	**32x48**	**3.072**	**316**	32	**98.304**
XCV400E	**40x60**	**4.800**	**404**	40	**153.600**
XCV600E	**48x72**	**6.912**	**512**	72	**221.184**
XCV1000E	**64x96**	**12.288**	512	96	**393.116**
XCV1600E	72x108	15.552	660	144	497.664
XCV2000E	80x120	19.200	804	160	614.400
XCV2600E	92x138	25.392	804	184	812.544
XCV3200E	104x156	32.448	804	208	1.038.336

3.2.3 Arquitectura

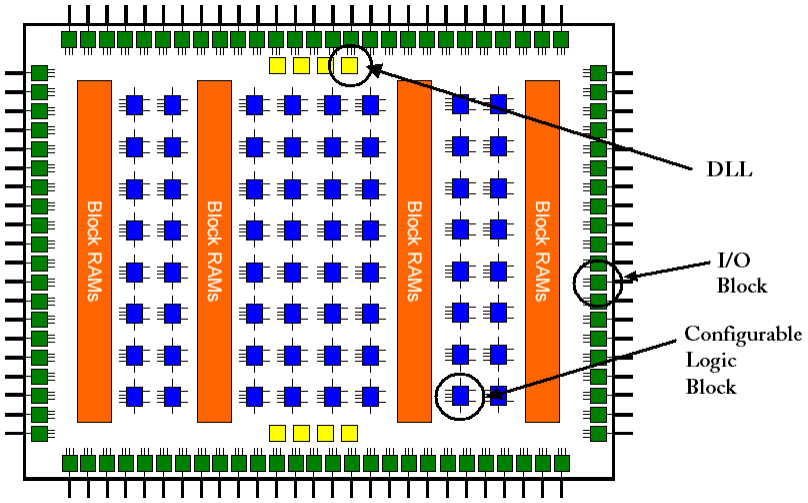


Ilustración 52: Arquitectura Virtex-E

- CLBs, proporcionan los bloques funcionales para constituir la lógica
- IOBs, proporciona el interfaz entre los pines y los CLBs
- Block RAMs, memorias dedicadas de doble puerto de 4096 bits
- DLLs, compensación en la propagación del reloj

3.3 Virtex-E EM 1.8V

3.3.1 Principales características

- Virtex-E Extended Memory, Posee las mismas características que Virtex-E, no obstante Virtex-E EM utiliza bloques de memoria de mayor tamaño.

3.3.2 Dispositivos pertenecientes

Dispositivo	CLB Array	Slices	Max I/O	BlockRAM 4kb	Distributed RAM bits
XCV405E	40x60	4.800	404	140	153.600
XCV812E	56x84	9.408	556	280	301.056

3.4 Virtex-II

3.4.1 Principales características

- Densidades desde 40k puertas hasta 8 Millones de puertas.
- Rendimiento de sistema superior a 420MHz
  - I/O superior a 840Mbps
- Compatible con PCI y PCI-X
- Soporta 25 intefaces estándar de alto rendimiento
- Más de 12 DCMs
- Sistema de memoria jerárquico
- Bloques con multiplicadores de 18 bits
- Arquitectura flexible
  - acarreamiento rápido lógico en cascacada look-ahead
  - ayuda del multiplicador
  - encadenamieno en cascada a la entrada de funciones
  - cascadable16-bit shift registers
  - interconexiones internas con soporte de tri-estados
- Configuración del sistema basada en SRAM
  - reconfiguración parcial
  - encriptación 3DES del bitstream
- Proceso metálico de 8 capas 0.15µm

3.4.2 Dispositivos pertenecientes

Dispositivo	CLB Array	Slices	Max I/O	BlockRAM 4kb	Multiplier Blocks	Distributed RAM bits
XC2V40	8x8	256	88	4		8.192
XC2V80	16x8	512	120	8		16.384
XC2V250	24x16	1.536	200	24		49.152
XC2V500	32x24	3.072	264	32		98.304
XC2V1000	40x32	5.120	423	40		163.840
XC2V1500	48x40	7.680	528	48		245.760
XC2V2000	56x48	10.752	624	56		344.064
XC2V3000	64x56	14.336	720	96		458.752
XC2V4000	80x72	23.040	912	120		737.280
XC2V6000	96x88	33.792	1.024	144		1.081.344
XC2V8000	112x104	46.592	1.024	168		1.490.944

3.4.3 Arquitectura

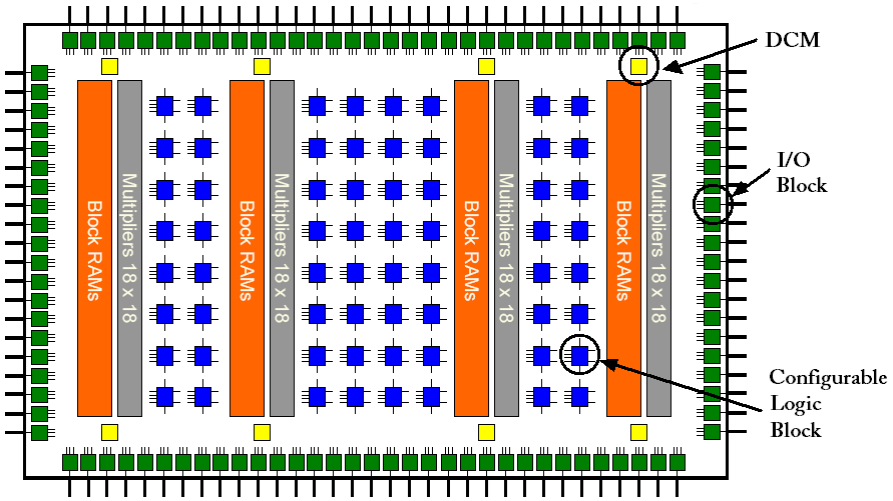


Ilustración 53: Arquitectura Virtex-II

La lógica configurable interna incluye cuatro elementos importantes organizados en array:

- Los bloques de lógica configurables (**CLBs**) proporcionan los elementos funcionales para lógica combinatoria y síncrona, incluyendo elementos básicos de almacenamiento. Los buffers triestado (BUFTs) asocian a cada elemento CLB mediante líneas horizontales de encaminamiento.

- Cada uno de los bloques **SelectRAM** (BlockRAMs) proporcionan módulos de memoria RAM de doble puerto de 18 Kbits. Una memoria de múltiple puerto es aquella que en el mismo instante de tiempo puede efectuar operaciones de lectura y escritura en bloques distintos de memoria. Por lo tanto una memoria de doble puerto puede realizar una operación de lectura mientras realiza otra de escritura. En cambio las memorias de puerto único o simple, solo pueden realizar en el mismo instante de tiempo o bien una operación de lectura o bien una de escritura.

- Cada uno de los bloques **Multiplicator** proporcionan multiplicadores dedicados de 18 bits x 18 bits.

- Cada uno de los bloques de **DCM** (encargado del reloj Digital, Digital Clock Manager), proporcionan el auto-calibrado de la señal, compensación del retardo en la expansión en la señal de reloj, multiplicación y división de la frecuencia del reloj, estrechamiento o estiramiento de la fase del reloj.

3.4.3.1 IOBs

Los bloques de E/S de Virtex-II™ (IOBs) están en grupos de dos o cuatro. Cada IOB puede ser usado como entrada o salida indistintamente. Se pueden utilizar dos bloques IOB para obtener un par diferencial (Differential Pair), este par diferencial esta conectado a la misma matriz encaminadora (switch matrix).

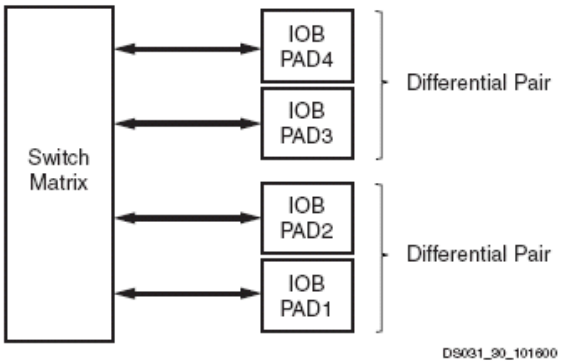


Ilustración 54: bloques E/S (IOBs) en Virtex-II, matriz encaminadora

Un bloque IOB esta formado por 3 sub-bloques:

- Un bloque para la entrada (Input)
- Un bloque para la salida (Output)
- Un bloque de direccionamiento triestado (3-State)

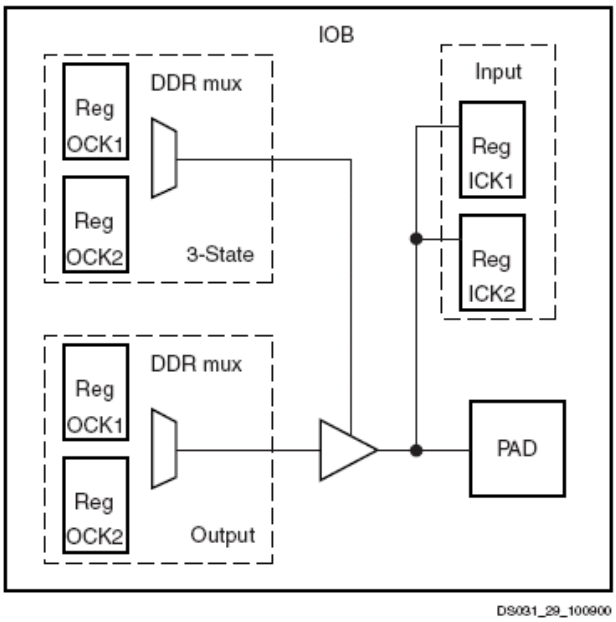


Ilustración 55: bloques E/S (IOBs) en Virtex-II, diagrama interno de un IOB

Cada uno de los IOBs tiene asociado 6 registros. Dos para cada subbloque: dos registros para la entrada, dos registros para la salida y dos registros para el direccionamiento triestado. Cada uno de los registros puede estar configurado como un flip-flop tipo D edge-triggered, o como un latch level-sensitive.

Mediante el uso de un multiplexor DDR (Double data Regiter) se pueden usar 1 o 2 registros para la entrada, la salida o para el direccionamiento triestado. Para seleccionar un registro se utiliza el reloj como señal de control para sincronizar los dos registros. Puede usarse para propagar una copia del reloj, de forma que datos y reloj tengan un retardo idéntico.

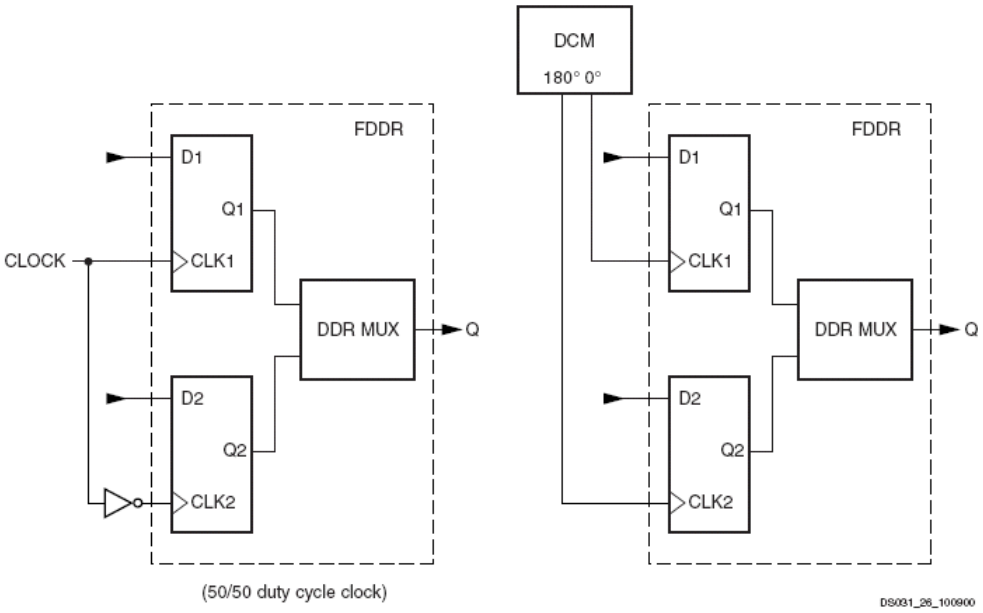


Ilustración 56: bloques E/S (IOBs) en Virtex-II, Registros DDR

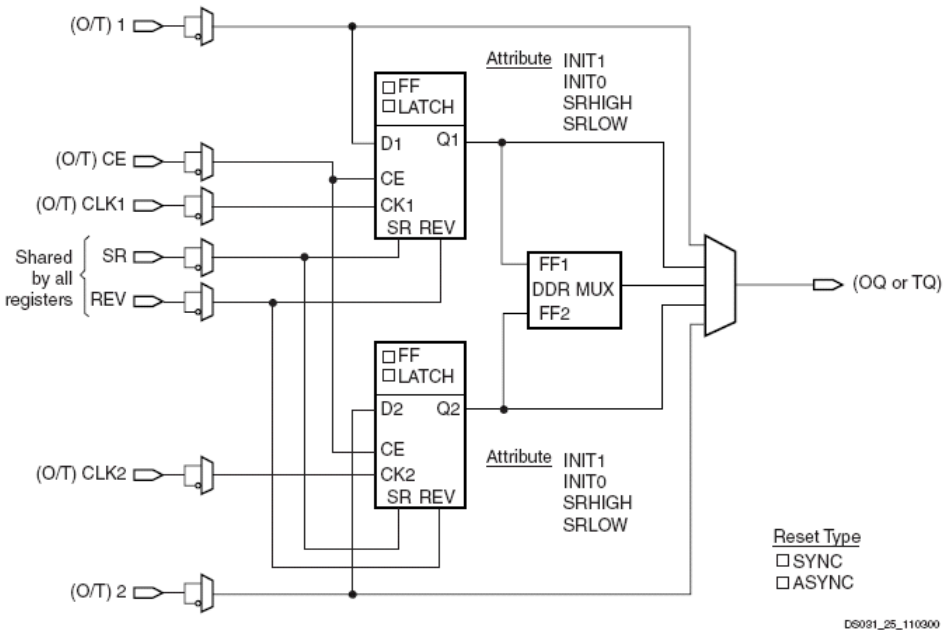


Ilustración 57: bloques E/S (IOBs) en Virtex-II, configuración registro/latch

Cada uno de los pads de entrada/salida tiene una resistencia opcional de pull-up o pull-down (rango de 10 a 60 KΩ), para seleccionar el modo de funcionamiento, ya sea LVTTL, LVCMOS o PCI. VCCO debe estar alimentado a 3.3V (3.0V a 3.6V). El diodo de protección (Clamp diode) siempre estará presente, aún cuando el pad no esté alimentado.

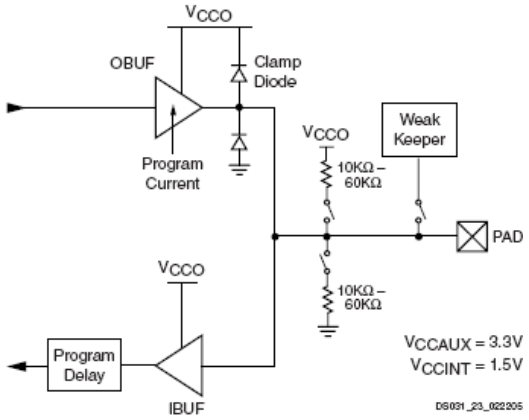


Ilustración 58: bloques E/S (IOBs) en Virtex-II, selección estándar LVTTL, LVCMOS o PCI

SelectI/O-Ultra	Programmable Current (Worst-Case Guaranteed Minimum)						
	2 mA	4 mA	6 mA	8 mA	12 mA	16 mA	24 mA
LVTTL							
LVCMOS33							
LVCMOS25							
LVCMOS18							n/a
LVCMOS15							n/a

FIG: LVTTL and LVCMOS Programmable Currents (Sink and Source)

3.4.3.2 Configurable Logic Blocks (CLBs)

- **CLBs**:
  - Los bloques de lógica configurables (CLBs) de Virtex-II se organizan en array y se utilizan para construir diseños lógicos combinatorios o síncronos.
- **Switch Matrix:**
  - Cada uno de los CLB se conecta a la matriz encaminadota (switch matrix) para tener acceso a la matriz general de encaminamiento.
- **Slices:**
  - Cada CLB incluye 4 slices interconectados entre sí por conexiones rápidas.
- **COUT**:
  - Existe lógica e interconexionado especial para aceleración de carry
  - 2 carrys por CLB

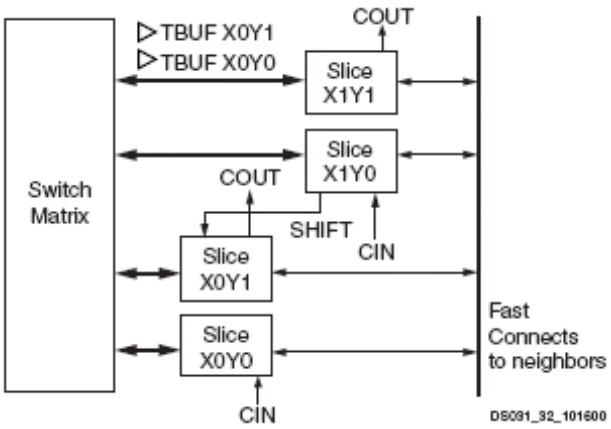


Ilustración 59: bloques Lógicos configurables (CLBs) en Virtex-II

Un CLB contiene 4 bloques similares (Slices X0Y0, X0Y1, X1Y0 X1Y0), con regeneración local rápida dentro del CLB. Los cuatro bloques (Slices) están agrupados en dos columnas de dos bloques (Slices X0Y0 con X0Y1, y X1Y0, X1Y1).

- Cadenas lógicas independientes (X1Y1 y X0Y0)
- Cadena en común de intercambio (X0Y1 y X1Y0).

Cada bloque (Slice) incluye:

- 2 generadores de funciones de 4 entradas.
- Acarreo lógico.
- Puertas de la lógica aritmética.
- Multiplexores.
- 2 registros de almacenamiento.

Cada generador de funciones de 4 entradas se puede programar como:

- LUT G, F: 1 tabla de verdad (LUT) de 4 entradas.
- RAM16: 16 bits de memoria distribuida SelectRAM .
- SRL16: 1 registro variable de desplazamiento de 16 bits.

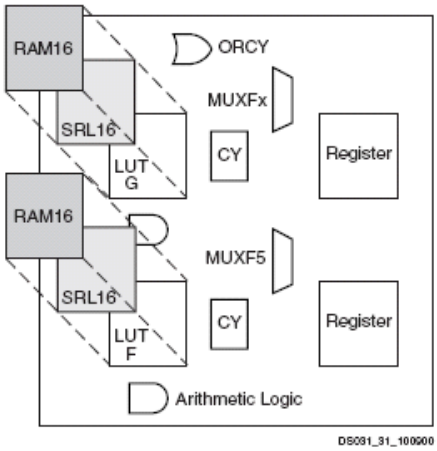


Ilustración 60: CLBs en Virtex-II, configuración de un Slice

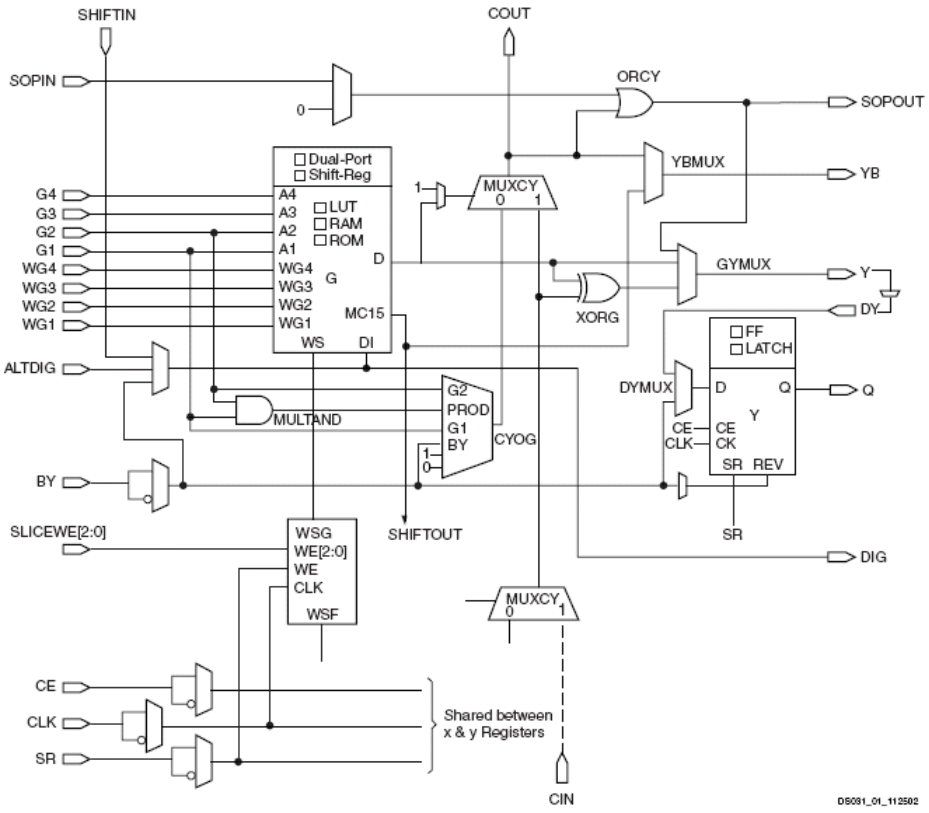


Ilustración 61: CLBs en Virtex-II, un Slice (1 de dos)

3.4.3.3 Tablas de verdad (Look-Up Table, LUT)

Los generadores de funciones de Virtex-II se pueden implementar como LUTs de 4 entradas. Se proporcionan cuatro entradas diferentes para cada uno de los generadores de funciones F y G. Cada uno, F y G, es capaz de poner cualquier función boleana definida para cuatro entradas en tiempo de ejecución. El retraso en la propagación es independiente de la función puesta en ejecución. Las señales de los generadores de funciones pueden salir del bloque Slice (Salida X o Y), pueden entrar en la puerta dedicada XOR, o en la entrada del multiplexor con acarreo lógico, o alimentar la entrada del elemento del almacenamiento (registro) D, o ir a MUXF5.

Además del LUTs básico, el bloque Slice de Virtex-II contiene la lógica (los multiplexores MUXF5 y MUXFX) suficiente para proporcionar funciones de cinco, seis, siete u ocho entradas. MUXFX puede ser MUXF6, MUXF7 o MUXF8 según el uso de bloque Slice en el CLB. Las funciones pueden como máximo hasta nueve entradas (multiplexor MUXF5).

3.4.3.3.1 Register/Latch

Los registros en una bloque Slice de Virtex-II se pueden configurar como un flip-flop edge-triggered tipo-D o como level-sensitive latches. La entrada D se puede controlar:

- Directamente por X o Y
- Por la salida de DX o DY
- Por el resultado del generador de funciones BX o BY.

La señal de reloj de enabled (CE) se activa a nivel alto por defecto.

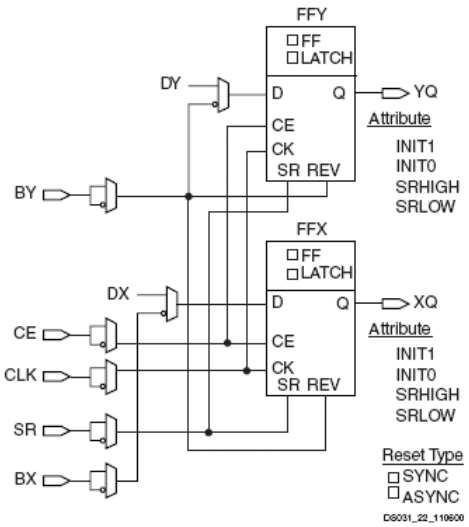


Ilustración 62: CLBs en Virtex-II, Configuración registro/latch

Además de las señales de reloj (CK) y permitir las señales del reloj (CE), cada bloque slice posee señales de reajuste (entradas SR y BY). SR realiza un SET con el valor especificado por SRHIGH o SRLOW. SRHIGH establece a "1" y

SRLOW establece a “0”. Cuando se utiliza la entrada SR y como segunda entrada BY se fuerza el establecimiento del estado opuesto, o lo que es lo mismo, del complementario de SRHIGH o SRLOW, “0” y “1” respectivamente. La condición de RESET es predominante sobre la condición SET. El estado inicial después de la configuración o del estado inicial se asocia separadamente con INIT0 e INIT1. Por defecto, SRLOW es establecido por INIT0, y SRHIGH es establecido por INIT1. Para cada bloque Slice, SET y RESET puede ser fijado para trabajar en modo síncrono o asíncrono. Los dispositivos de Virtex-II también tienen la capacidad de establecer INIT0 e INIT1 independientes de SRHIGH y de SRLOW. Las señales de control del reloj (CLK), reloj permitido (CE) y el set/reset (SR) son comunes a ambos elementos del almacenamiento en uno bloque Slice. Todas las señales de control tienen polaridades independientes. Cualquier inversor colocado en una entrada de control se absorbe automáticamente.

La funcionalidad de un registro set/resert puede configurarse:

- No set or reset
- Synchronous set
- Synchronous reset
- Synchronous set and reset
- Asynchronous set (preset)
- Asynchronous reset (clear)
- Asynchronous set and reset (preset and clear)

La señal de reset síncrona tiene preferencia a la señal síncrona de set.
La señal de clear asíncrona tiene preferencia a la señal asíncrona de preset.

3.4.3.3.2 Memoria distribuida SelectRam

Cada uno de los generadores de funciones "LUT", (cada slice continen 2 LUTs) puede implementar una memoria RAM síncrona de 16 x 1-bit, este elemento recibe el nombre de SelectRAM. Los elementos SelectRAM se pueden configurar dentro de un bloque CLB de los siguientes modos:

- Memoria de un puerto (S=Single-Port):
  - **RAM 16x1S:**
    - 16 x 8 bit RAM (1 LUT)
  - **RAM 32x4S:**
    - 32 x 4 bit RAM (2 LUT)
  - **RAM 64x2S:**
    - 64 x 2 bit RAM (4 LUT)
  - **RAM 128x1S:**
    - 128 x 1 bit RAM (8 LUT)

- Memoria de doble puerto (D=Double-Port):
  - **RAM 16x4D:**
    - 16 x 4 bit RAM (2 LUT)
  - **RAM 32x2D:**
    - 32 x 2 bit RAM (4 LUT)
  - **RAM 64x1D:**
    - 64 x 1 bit RAM (8 LUT)

Para las configuraciones de memoria RAM de puerto simple (Single-Port) de distributed SelectRAM, tienen una única dirección para escritura síncrona y lectura asíncrona, compartiendo las mismas lineas de bus de dirección.

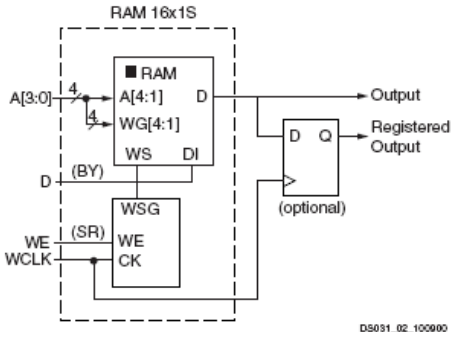


Ilustración 63: Distributed SelectRAM (RAM 16x1S)

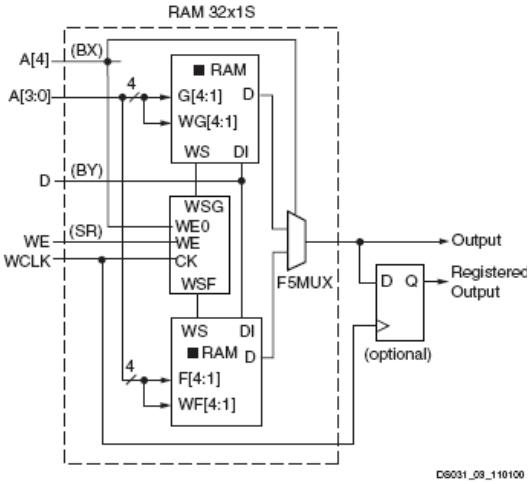


Ilustración 64: Single-Port Distributed Select-RAM (RAM32x1S)

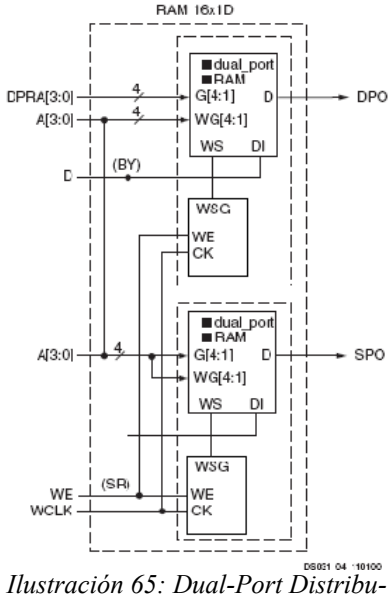


Ilustración 65: Dual-Port Distributed SelectRAM (RAM16x1D)

Para las configuraciones de memoria RAM de doble puerto (Double-Port) de distributed SelectRAM, hay un puerto para escritura síncrona y lectura asíncrona, y otro puerto para lectura asíncrona. El generador de funciones LUT, tiene lineas de direcciones separadas para la lectura (A1, A2, A3, A4) y escritura (WG1/WF1, WG2/WF2, WG3/WF3, WG4/WF4).

3.4.3.3.3 Registros de desplazamiento

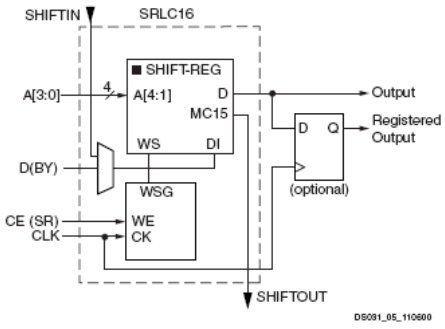
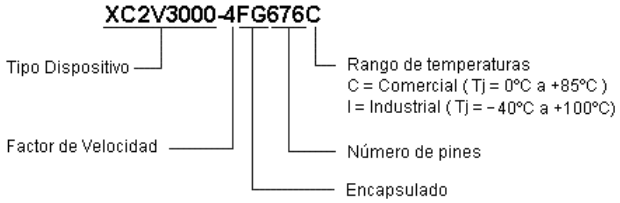


Ilustración 66: Configuración como registro de desplazamiento

3.5 XC2V3000-4FG676C

Se analizará la FPGA de Xilinx XC2V3000-4FG676C como ejemplo de análisis de una FPGA. El Part-Number o código identificador de las FPGAs de Xilinx nos proporciona la siguiente información:



Las principales características de esta FPGA son:

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V3000	3M	64 X 56	14,336	448	96	96	1,728	12	720

Los principales elementos constituyentes de esta FPGA son:

- DCM
- IOB, Bloque de entrada/salida
  - Accesibles al usuario
- CLB, Bloque de lógico configurable
- Multiplier, Multiplicador
- Block SelectRAM
- Global Clock Mux

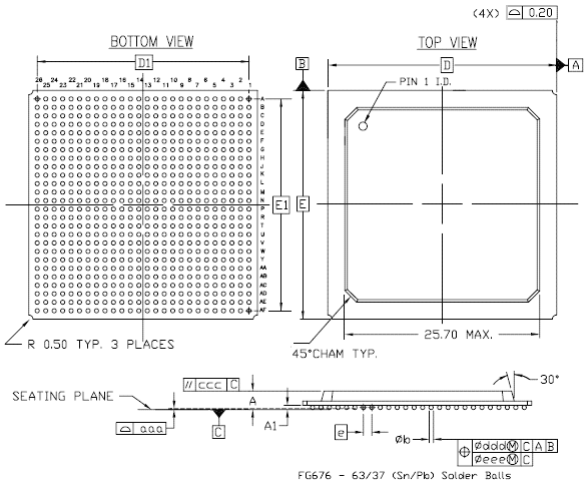


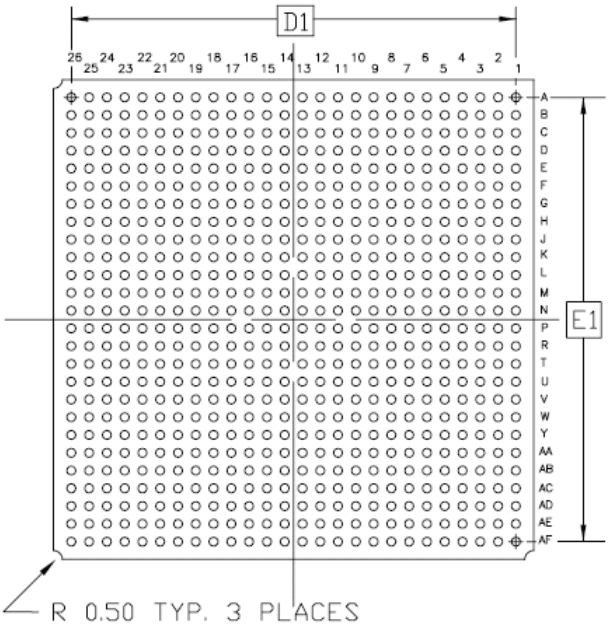
Ilustración 67: Encapsulado FG676

El dispositivo XC2V3000-4FG676 hace referencia a una FPGA del tipo XilinxVirtex-II de 3 millones de puestas lógicas, en encapsulado FG676:

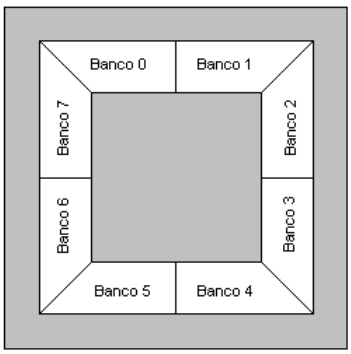
3.5.1 XC2V3000-4FG676C: Bloques Entrada/Salida (IOBs)

La FPGA XC2V3000 se vende en diferentes encapsulados. El número de IOBs depende del encapsulado, el máximo para esta arquitectura es de 720. La localización de los pines va determinado por el par fila (A…AF, no existe la I, O, Q, S, X y Z) columna (1…26).

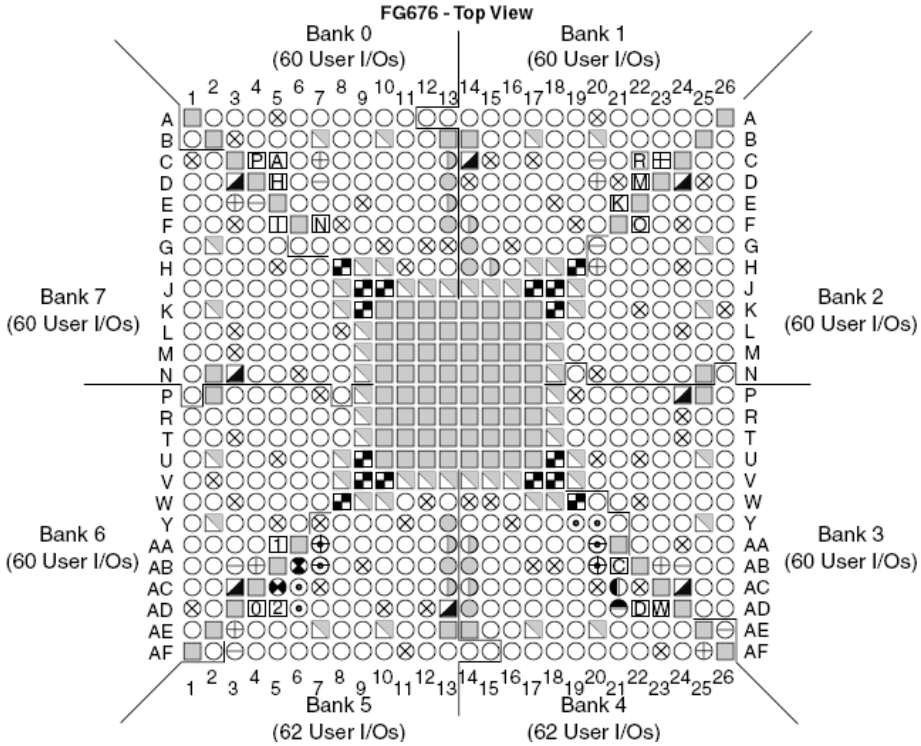
Dispositivo	Número IOBs
XCV3000-4FG676	484



Los Bloques de Entrada/Salida (IOBs) están divididos en 8 bancos [0,7]:

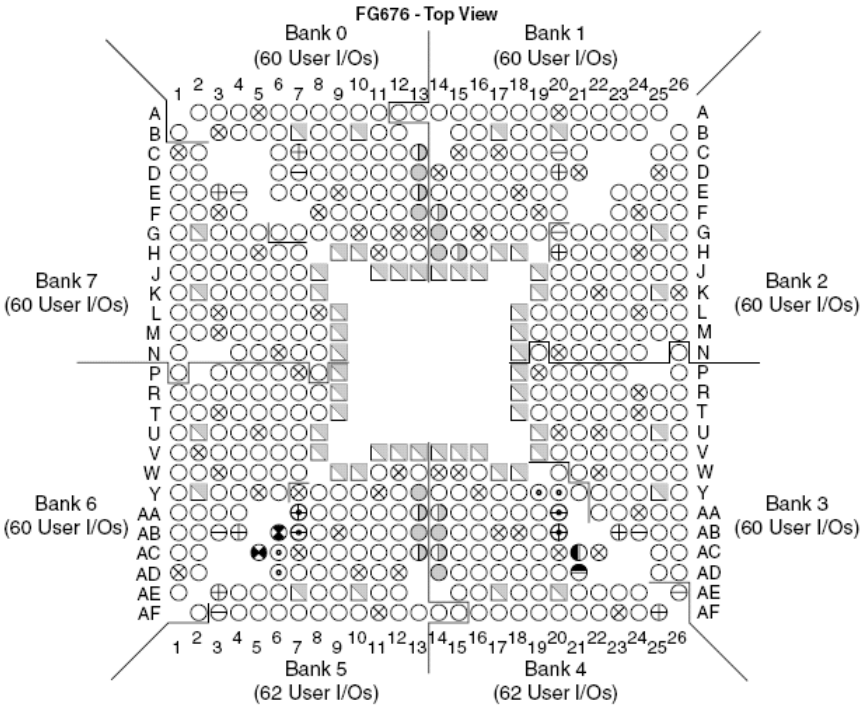


La parte central esta en blanco, ya que esta zona de la FPGA está reservada a pines con una función determinada. Los pines asociados a estos bancos son:



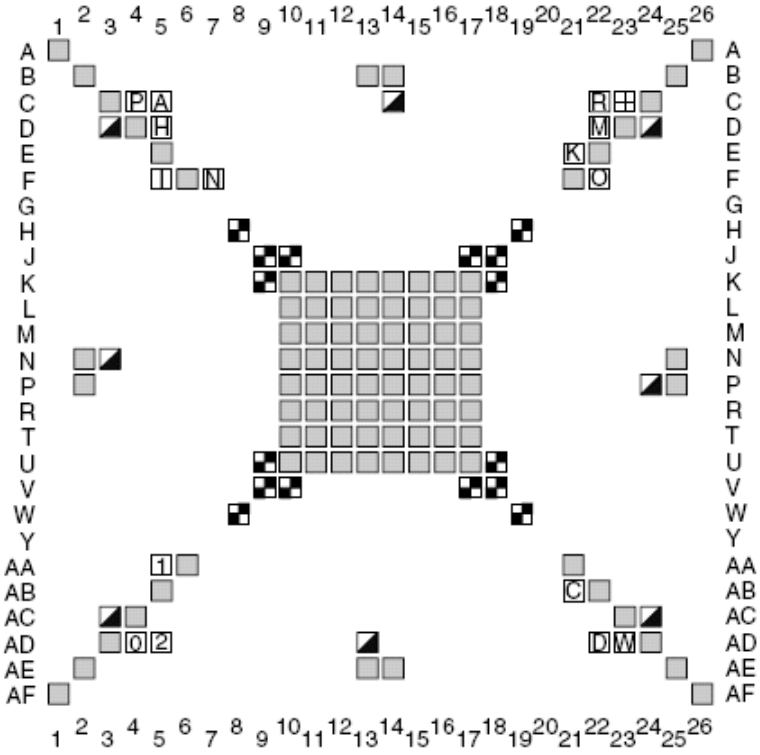
User I/O Pins	Dedicated Pins	
○ IO_LXXY_#	C CCLK	N DXN
Dual-Purpose Pins:	P PROG_B	A DXP
⊙ DIN/D0-D7	D DONE	⊞ VBATT
◓ CS_B	2 1 0 M2, M1, M0	R RSVD
◓ RDWR_B	H HSWAP_EN	▨ VCCO
◓ BUSY/DOUT	K TCK	◢ VCCAUX
◐ INIT_B	I TDI	▞ VCCINT
● GCLKx (P)	O TDO	▩ GND
◑ GCLKx (S)	M TMS	n NO CONNECT
⊖ VRP	W PWRDWN_B	
⊕ VRN		
⊗ VREF		
Triple-Purpose Pins:		
⊖ D2, D4/ALT_VRP		
⊕ D3, D5/ALT_VRN		

Los pines accesibles al usuario son:



User I/O Pins	Dedicated Pins	
IO_LXXY_#		
Dual-Purpose Pins:		
DIN/D0-D7		
CS_B		
RDWR_B		VCCO
BUSY/DOUT		
INIT_B		
GCLKx (P)		
GCLKx (S)		
VRP		
VRN		
VREF		
Triple-Purpose Pins:		
D2, D4/ALT_VRP		
D3, D5/ALT_VRN		

Los pines dedicados son:



User I/O Pins	Dedicated Pins	
	C CCLK	N DXN
	P PROG_B	A DXP
	D DONE	VBATT
	2 1 0 M2, M1, M0	R RSVD
	H HSWAP_EN	
	K TCK	VCCAUX
	I TDI	VCCINT
	O TDO	GND
	M TMS	n NO CONNECT
	W PWRDWN_B	

4 Placa de desarrollo RC203 de Celoxica

Este capítulo presenta los elementos que forman la placa de desarrollo RC203 de Celoxica. Se ha elegido esta placa por el número de periféricos que incorpora, y al estar todos conectados con la FPGA, una Virtex-II con capacidad suficiente para controlarlos. Debido a su tamaño, cantidad de memoria/numero de puertas, esta placa de desarrollo sirve para realizar aplicaciones no demasiado complejas, no obstante de gran variedad.

Para realizar el diseño con cualquier placa de desarrollo se necesita conocer de antemano los periféricos que incorpora y los pines que tienen asignados para su interfaz. Es por este motivo por el que en este capítulo se detallan para cada uno de los periféricos que incorpora la placa de desarrollo, sus esquemas eléctricos, los pines asignados a la FPGA, y su funcionalidad. Ya que esta información se consultará para realizar la programación de dicho dispositivo.

4.1 Componentes

- Virtex-II 2V3000-4
- Ethernet MAC/PHY a 10/100baseT
- Bancos de SRAM ZBT SRAM proporcionando un total de 8-MB
- Soporte vídeo:
  - Entrada/Salida compuesta de vídeo
  - Entrada/Salida S-Vídeo
  - Salida VGA
  - Entrada Camera (El conector proporciona la alimentación de la cámara)
- Pantalla táctil
- Cámara a color
- Audio compatible con AC'97:
  - Entrada micrófono
  - Entrada de línea (estéreo)
  - Line/Headphone out (estéreo)
- Auriculares y micrófono
- Ratón
- Conector para la memoria Flash SmartMedia para almacenar los archivos BIS
- CPLD for configuration/reconfiguration and SmartMedia management
  - Power-on load from SmartMedia
  - Load when SmartMedia installed
  - Reconfigure on demand from FPGA
- Conector JTAG
- Puerto paralelo y cable, para descargar el archive BIT, y también para comunicar la FPGA con el PC
- RS-232
- Modulo Bluetooth
- Conectores de teclado y ratón PS/2
- 2 displays de 7-segmentos
- 2 LEDs azules
- 2 interruptores
- 50 pines de expansión:
  - 33 pines digitales de E/S
  - 3 pines de alimentación (+12V, +5V, +3.3V)
  - 2 pines de reloj

- Tarjeta SmartMedia de 16-MB
- Alimentador universal 110/240V

4.2 Dispositivos

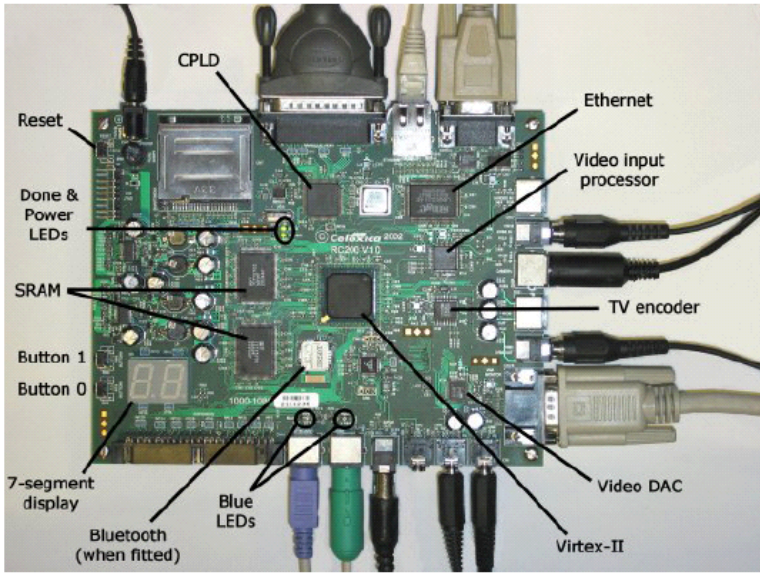


Ilustración 68: Dispositivos RC203-E

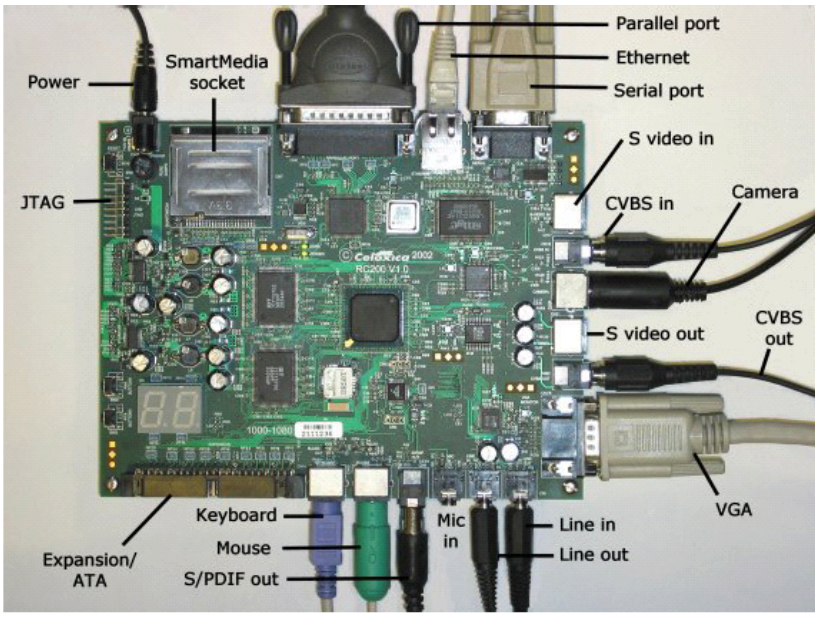


Ilustración 69: Conectores RC203-E

4.3 Esquemas y pines de interconexión

4.3.1 LEDs

La tarjeta RC203-E tiene dos LEDs azules que pueden ser controlados directamente por la FPGA. Los pines de los LEDs tienen que ser puestos a uno para encender los LEDS.

Pines del LED	Pines FPGA RC203
Blue0	L8
Blue1	M8

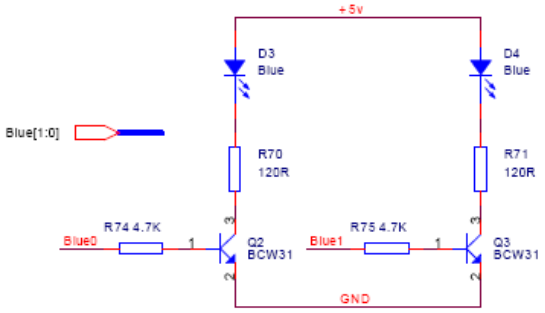
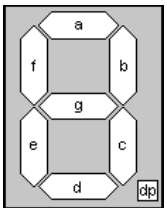
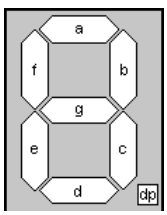


Ilustración 70: RC203-E, conexión leds

4.3.2 DISPLAY

La tarjeta RC203-E tiene dos displays de 7 segmentos que pueden ser controlados directamente por la FPGA.

Display 0			
	**7-segmentos**	**Segmento**	**FPGA RC203**
	A1	a	J5
	B1	b	K6
	C1	c	N5
	D1	d	N6
	E1	e	M5
	F1	f	H5
	G1	g	J6
	DP1	Punto decimal	N7
Display 1			
	**7-segmentos**	**Segmento**	**FPGA RC203**
	A2	a	L6
	B2	b	L5
	C2	c	K7
	D2	d	H7
	E2	e	N8
	F2	f	K5
	G2	g	J7
	DP2	Punto decimal	M6

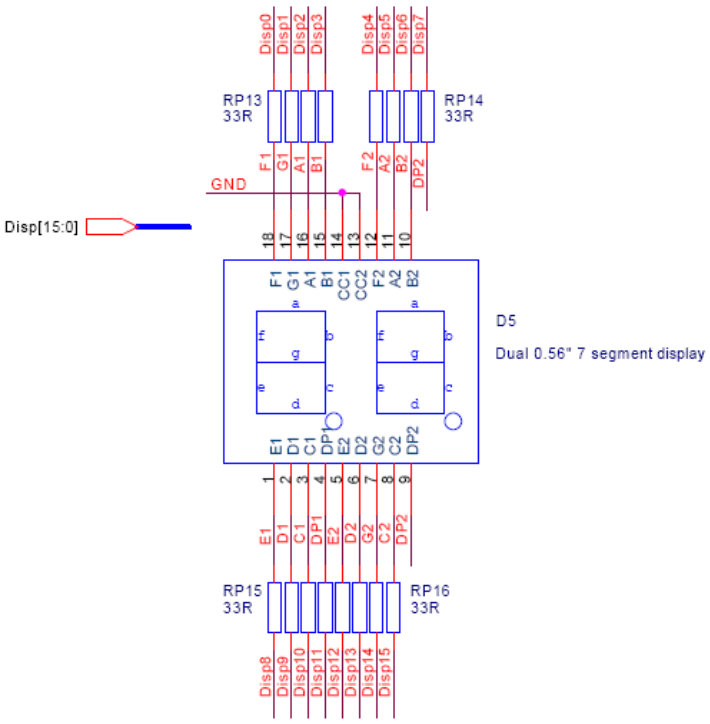


Ilustración 71: RC203-E, conexión display de 7 segmentos

4.3.3 RS-232

La tarjeta RC203-E contiene un puerto RS232, los pines de interconexión con la FPGA son los siguientes:

Descripción	Función	Pines FPGA RC203
Serial0	CTS (Clear To Send)	V21
Serial1	RxD (Receive data)	W22
Serial2	RTS (Ready To Send)	W21
Serial3	TxD (Transmit data)	Y22

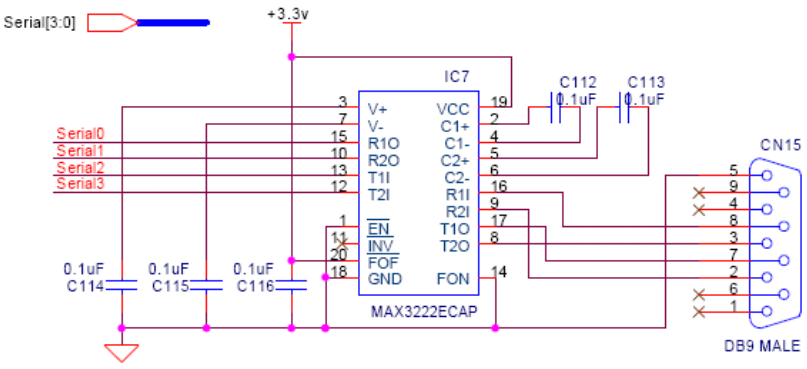


Ilustración 72: RC203-E, Conexión RS232

4.3.4 Audio

El chip Cirrus Logic CS4202 en un codec estéreo de audio compatible con AC'97 que incluye sonido surround y multicanal para aplicaciones para los PCs.

Pines audio codec	Función	Pines FPGA RC203
AC0	SDATA_OUT	AC7
AC1	BIT_CLK	AC8
AC2	SDATA_IN	AD8
AC3	SYNC	AC9
AC4	nRESET	AD9

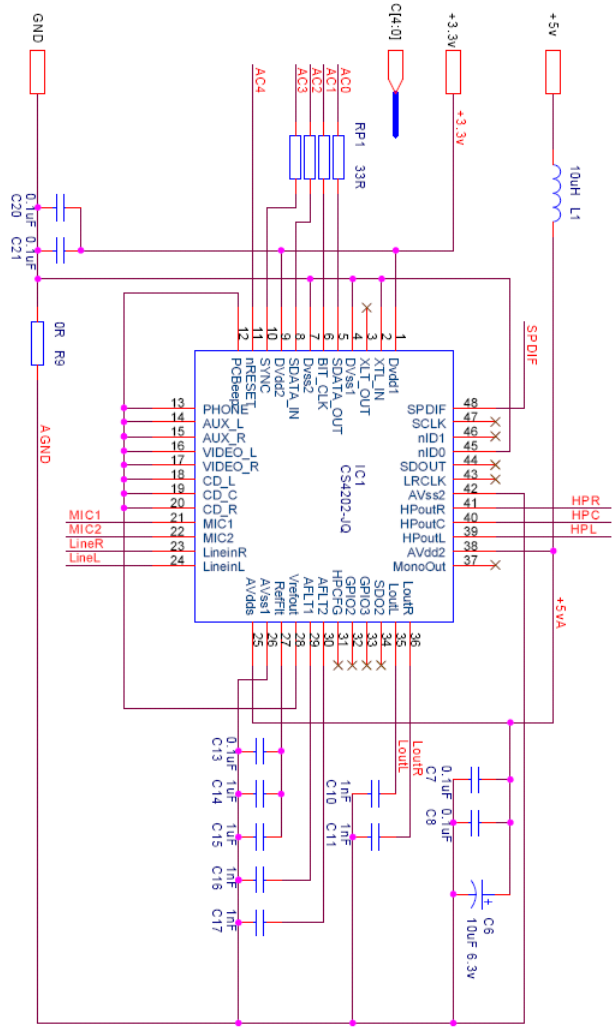


Ilustración 73: RC203-E, Conexión FPGA con CI Cirrus Logic CS4202

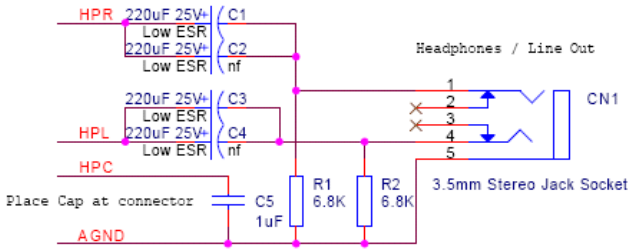


Ilustración 74: RC203-E, Conexión Line Out

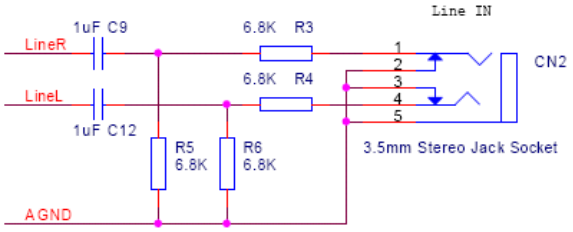


Ilustración 75: RC203-E, conexión Line IN

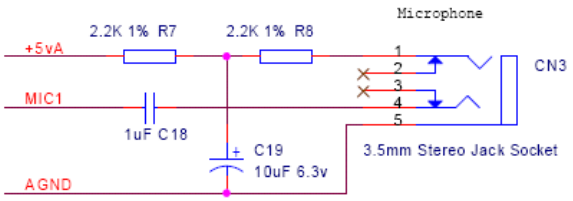


Ilustración 76: RC203-E, conexión microfono

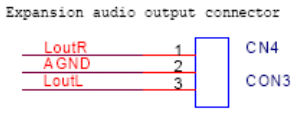


Ilustración 77: RC203-e, conector expansión de salida de audio

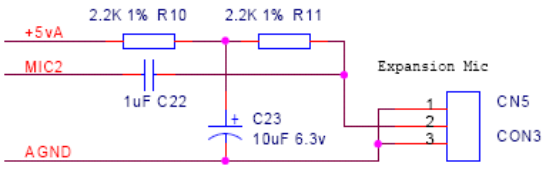


Ilustración 78: RC203-E, conector expansión de micrófono

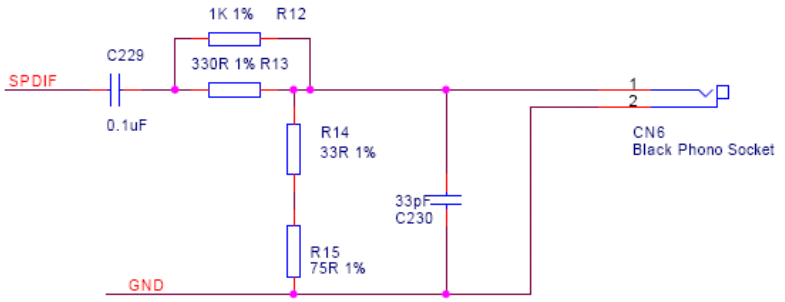


Ilustración 79: RC203-E, conector expansión de auriculares

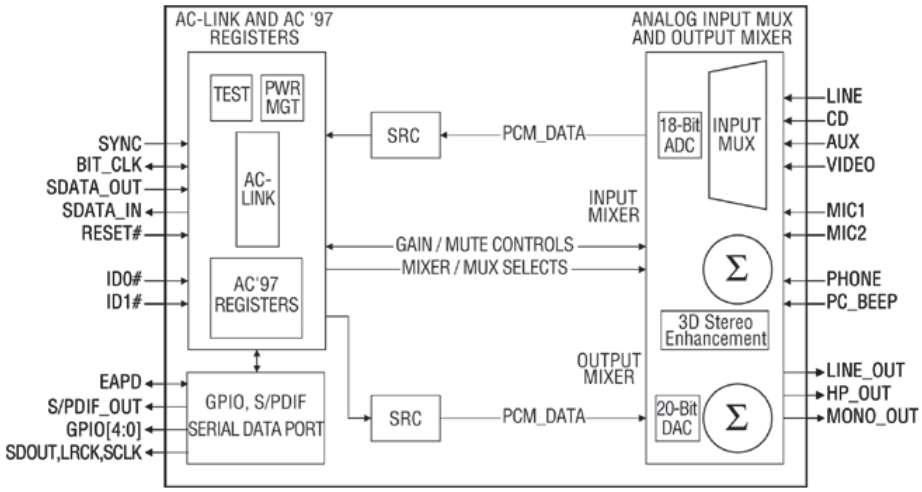


Ilustración 80: Diagrama CI Cirrus Logic CS4202

4.3.5 BlueTooth

El módulo de Bluetooth Mitsumi WML-C09 está conectado directamente a la FPGA. Los pines de interconexión co la FPGA son los siguientes:

Pines Bluetooth	Función	Pines FPGA RC203
BT0	Pin RX	AA15
BT1	Pin TX	AB15
BT2	Pin RTS	AA14
BT3	Pin CTS	Y14
BT4	Pin Reset	W14

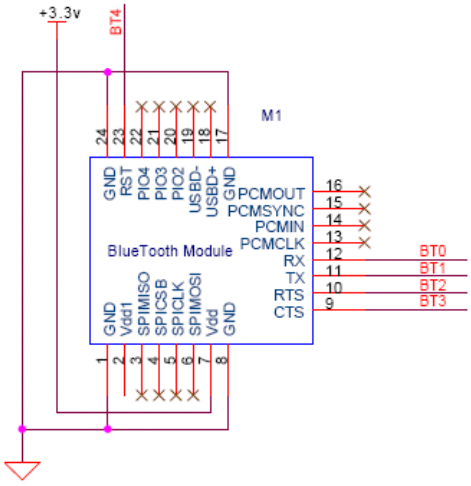


Ilustración 81: RC203-E, módulo de Blueto-oth Mitsumi WML-C09

4.3.6 Pantalla táctil

Los pines de interconexión de la FPGA RC203-E con la pantalla táctil son los siguientes:

Touch screen	Pines FPGA RC203
nPENIRQ	AB16
nCSTOUCH	AA16
SPI CLK	AB18
SPI DIN	AA17
SPI DOUT	AB17

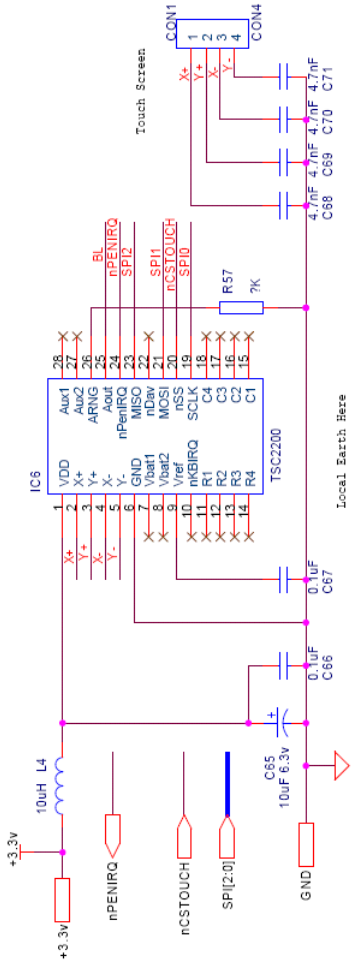


Ilustración 82: RC203-E, conexión con la pantalla táctil

4.3.7 LAN

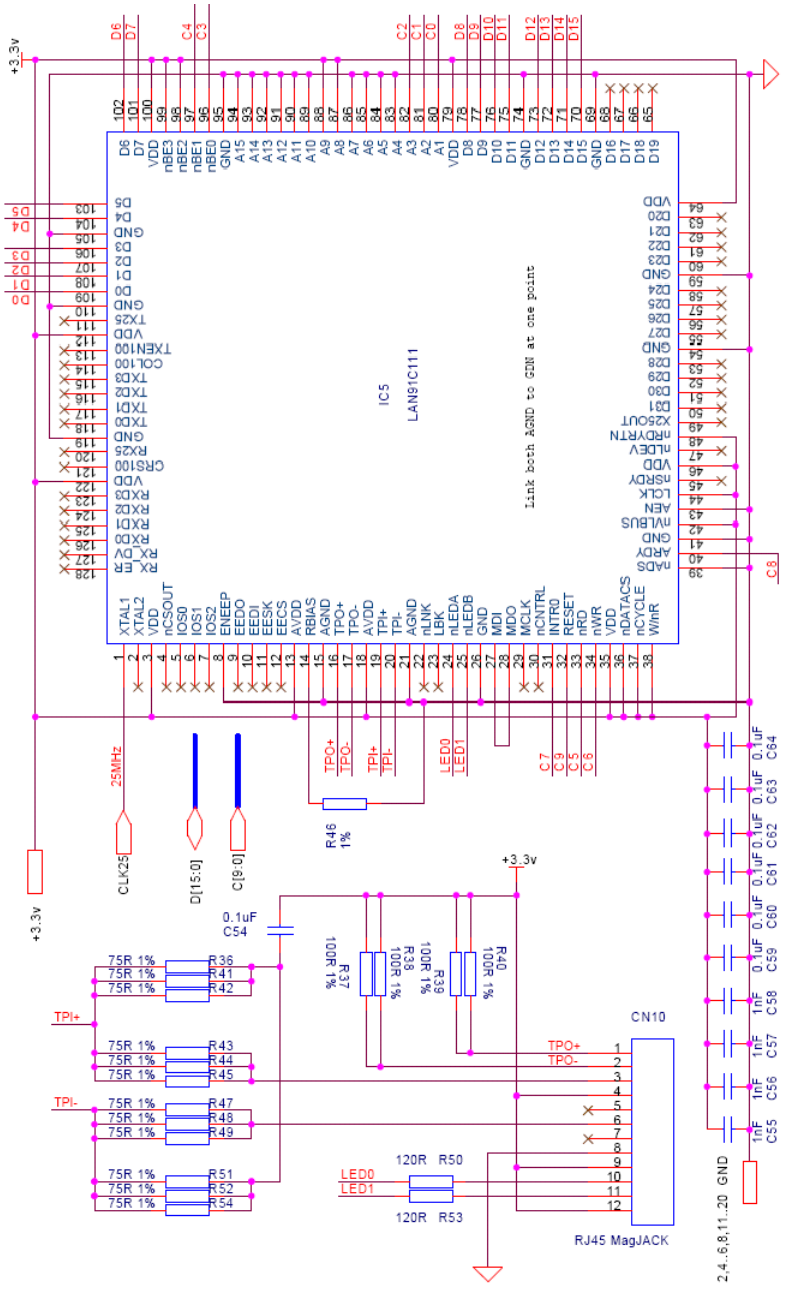


Ilustración 83: RC203-E, interconexión con LAN

Pines Ethernet	Descripción	Pines FPGA RC203
ED0 - ED15	Datos [15:0]	P23, R24, R23, T24, T23, U23, V24, V23, W24, W23, Y23, AA24, AA23, AB24, AB23, P19
EC0 - EC2	Dirección [2:0]	P20, P22, P21

EC3 y EC4	Not byte enable	R22, R21
EC5	No lectura	T22
EC6	No escritura	T21
EC7	Interrupción	U22
EC8	Asynchronous ready pin (Ardy)	U21
EC9	Reset	V22

4.3.8 PS/2: Mouse & Keyboard

Los pines de interconexión entre la FPGA con el ratón y el teclado son:

Pines PS/2	Descripción	Pines FPGA RC203
KM0	Ratón DATA	T7
KM1	Ratón CLK	U7
KM2	Teclado DATA	V7
KM3	Teclado CLK	W7

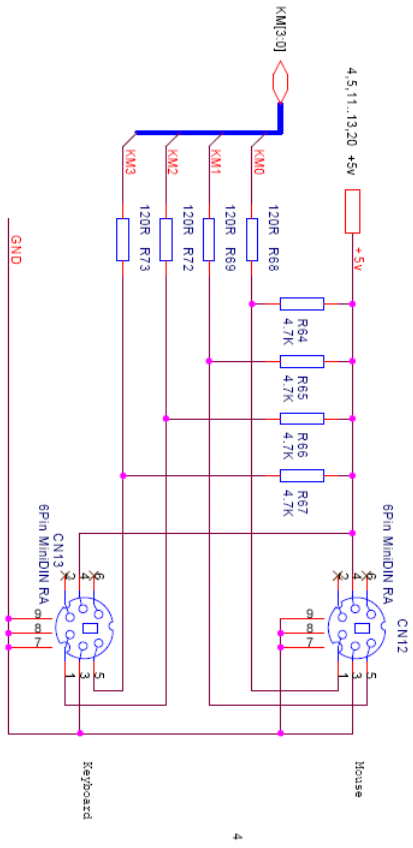


Ilustración 84: RC203-E, interconexión con PS2

4.3.9 Reloj externo

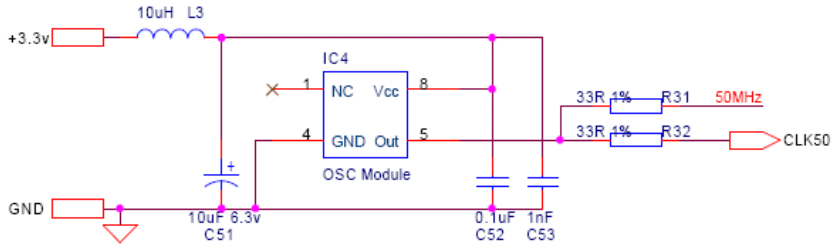


Ilustración 85: RC203-E, conexión reloj con FPGA

4.3.10 Reloj configurable

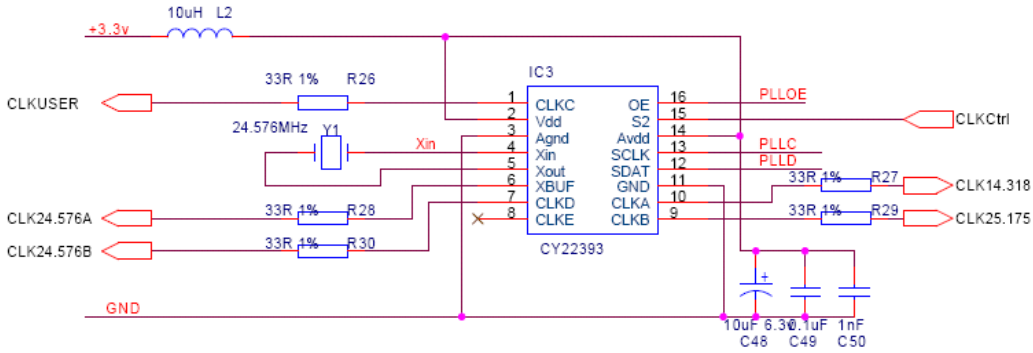


Ilustración 86: RC203-E, reloj configurable

Pines de creación de reloj	Descripción	Pines FPGA RC203
GCLK2P	CLKUSER. Reloj para gobernar la FPGA.	AB14
GCLK5P	Reloj a 24.576MHz. Reloj para gobernar la entrada de vídeo y sonido.	D13
GCLK6S	Reloj a 25.175MHz. Reloj para gobernar la salida VGA (640 x 480 a 60Hz).	E13
GCLK0P	Reloj de entrada de vídeo a 27MHz.	AD14
GCLK1P	Reloj de cuarzo a 50MHz. Reloj para gobernar el CPLD.	G14
GCLK7S	Expansión reloj 0	AC13

GCLK5S	Expansión reloj 1	AA13
	CLKCTRL	Y21

Frecuencias de reloj TV:
El generador de reloj proporciona frecuencias de 14.318MHz y 17.7MHz para el encoder RGB a PAL/NTSC. Se puede seleccionar un valor comprendido entre este rango mediante la señal CLKCTRL (pin 15 en el generador de reloj).

Reloj de la FPGA: CLKUSER
CLKUSER tiene por defecto el valor de 133MHz. Se puede cambiar el valor por defecto de CLKUSER programando el PLL desde la FPGA o desde el puerto paralelo.

4.3.11 Disco duro

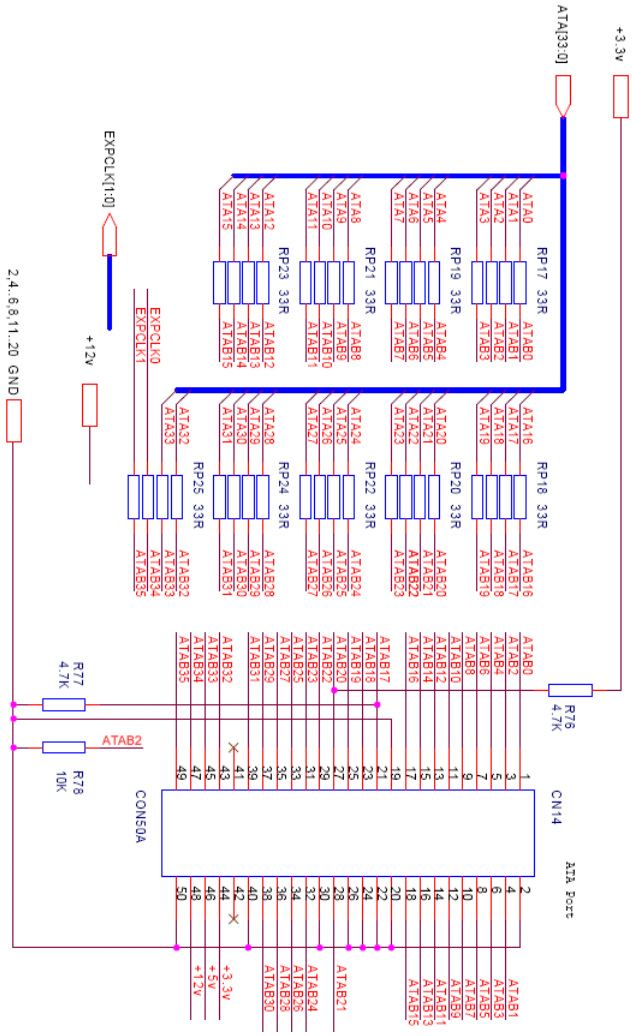


Ilustración 87: RC203-E, Conexión con un disco duro

Pin	Función ATA	Función del pin	Pines FPGA RC203
1	Reset	IO0	U4
2	GND	GND	-
3	D7	IO2	P4
4	D8	IO1	P3
5	D6	IO4	R4
6	D8	IO3	R3
7	D5	IO6	T4
8	D10	IO5	T3
9	D4	IO8	P6
10	D11	IO7	P5
11	D2	IO10	R6
12	D12	IO9	R5
13	D2	IO12	T5
14	D13	IO11	T6
15	D1	IO14	U6

Pin	Función ATA	Función del pin	Pines FPGA RC203
16	D14	IO13	U5
17	D0	IO16	V5
18	D15	IO15	V4
19	GND	GND	-
20	Keypin	Pin eliminado	-
21	DMARQ	IO17	V3
22	GND	GND	-
23	nDIOW	IO18	W3
24	GND	GND	-
25	nDIOR	IO19	V6
26	GND	GND	-
27	IORDY	IO20	W6
28	CSEL	IO21	Y5
29	nDMACK	IO22	Y6
30	GND	GND	-
31	INTRQ	IO23	AA3
32	Reservado	IO24	AA4
33	DA1	IO25	W4
34	nPDIAG	IO26	W5
35	DA0	IO27	R8
36	DA2	IO28	T8
37	nCS0	IO29	P7
38	nCS1	IO30	Y4
39	nDASP1	IO31	U3
40	GND	GND	-
41	Pin eliminado	Pin eliminado	-

42	Pin eliminado	Pin eliminado	-
43	IO32	IO32	Y3
44	+3.3v	+3.3v (máx. 0.5Amps)	-
45	IO33	IO33	R7
46	+5v	+5v (máx. 0.5Amps)	-
47	CLK0	CLK0	AC13
48	+12v	+12v (máx. 0.5Amps)	-
49	CLK1	CLK1	AA13
50	GND	GND	-

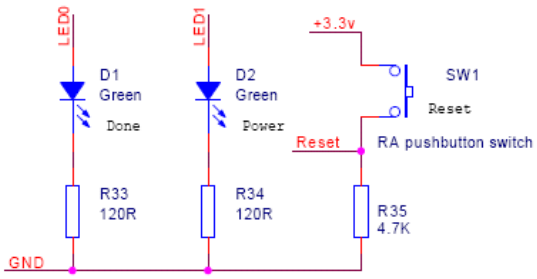
4.3.12 CPLD

La placa RC203-E tiene un CPLD XC95144XL 3.3V de Xilinx.
El CPLD está conectado a:

- FPGA
- Puerto Paralelo
- SmartMedia Flash RAM
- JTAG

El CPLD puede configurar la FPGA con los datos recibidos desde la memoria SmartMedia, o bien desde el puerto paralelo.

4.3.13 Resetear FPGA



4.3.14 Pulsadores

Hay dos pulsadores (Button0 y Button1), que cuando son pulsados proporcionan un valor alto en la entrada de la FPGA.

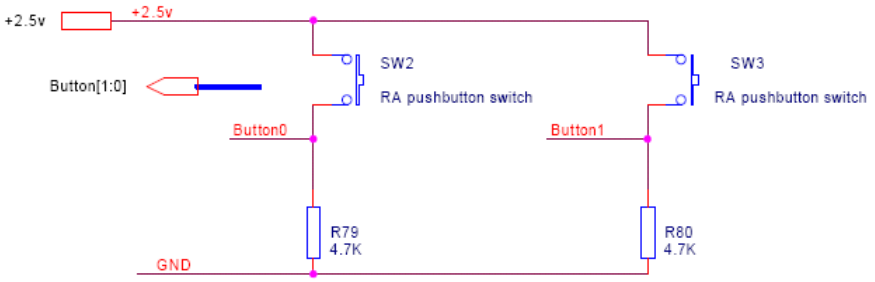


Ilustración 88: RC203-E, Conexión de los pulsadores

4.3.15Synchronous Static RAM

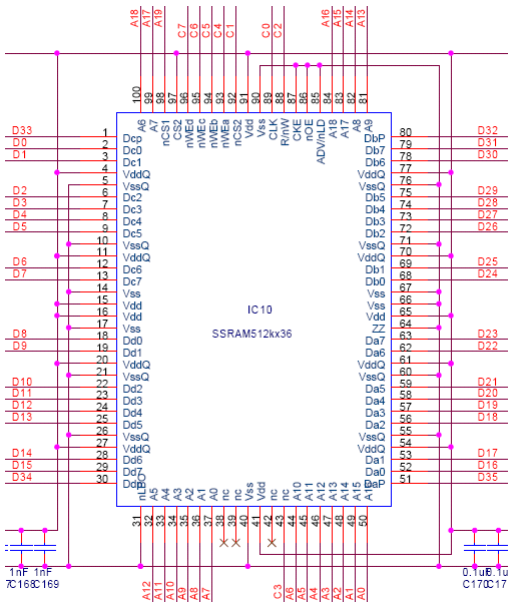


Ilustración 89: RC203-E, SDRAM módulo 1

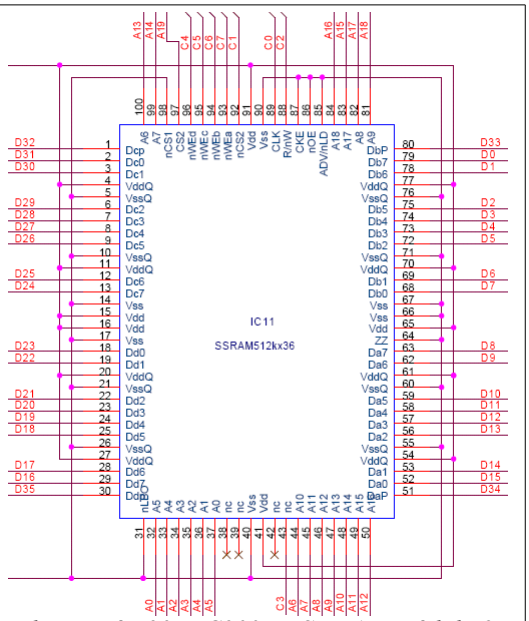


Ilustración 90: RC203-E, SDRAM módulo 2

FIG: Interconexión entre la FPGA y el banco 1 RAM

Pines SSRAM	Función	Pines FPGA RC203 (en orden ascendente)
S1D0 - S1D35	Datos [35:0]	F9, E9, F10, E10, F11, E11, F12, E12, G13, H13, G6, G7, G8, G9, G10, G11, G12, H11, H12, E4, E3, F4, F3, G4, H4, H3, J4, J3, K4, L4, L3, M4, M3, N4, G5, H6
S1A0 - S1A19	Dirección [19:0]	F19, E20, F20, H15, H16, G15, G16, G17, G18, G19, D6, C6, D7, D8, C8, D9, C9, D10, D11, C11
S1C0	Reloj	F13
S1C1	nCS2 (not Chip Select)	D112
S1C2	R/nW (Read not Write)	C12

S1C4 - S1C7	Not Byte Enable pins	F8, E8, E6, E7

FIG: Interconexión entre la FPGA y el banco 0 RAM

Pines SSRAM	Función	Pines FPGA RC203 (en orden ascendente)
S1D0 - S1D35	Datos [35:0]	M22, N21, N22, M20, N20, G20, H20, J20, K20, L20, L19, M19, D14, C15, D15, C16, D16, D17, C18, D18, C19, D19, D20, C21, D21, E14, F14, E15, F15, E16, F16, E17, F17, E18, F18, E19
S1A0 - S1A19	Dirección [19:0]	E23, E24, F23, F24, G23, H23, H24, J23, J24, K23, L23, L24, M23, M24, N24, N23, G21, G22, H21, H22
S1C0	Reloj	H14
S1C1	nCS2 (not Chip Select)	J21
S1C2	R/nW (Read not Write)	J22
S1C4 - S1C7	Not Byte Enable pins	L22, M21, K22, L21

4.3.16 Puerto paralelo

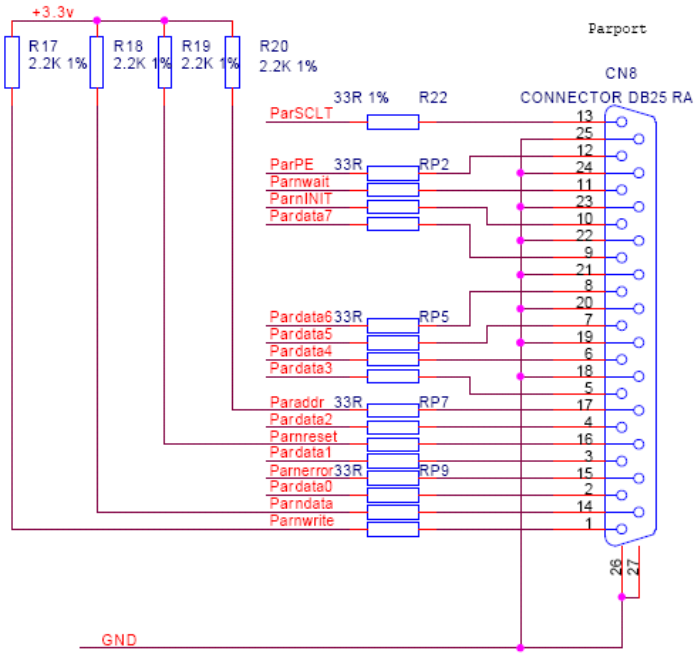


Ilustración 91: RC203-E, puerto paralelo

4.3.17 JTAG

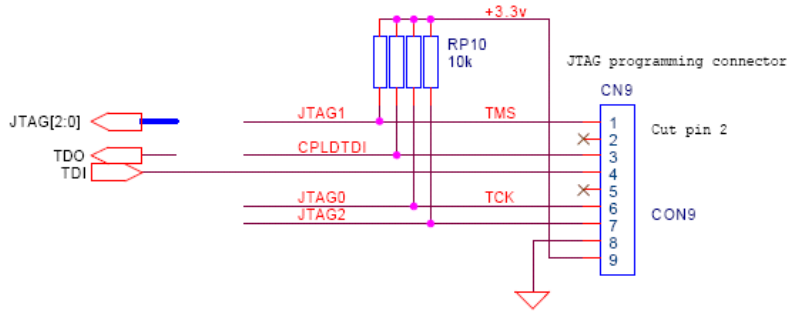


Ilustración 92: RC203-E, puerto JTAG

4.3.18 Entrada de Vídeo

La tarjeta RC203 posee un procesador de entrada de vídeo Philips SAA7113H , permitiendo la captura desde S Vídeo, CVBS y una Camera de vídeo.

La FPGA puede decodificar RGB:

- NTSC o PAL: utilizando el encoder AD725 RGB a NTSC/PAL
- Salida VGA utilizando el encoder ADV7123 RGB a VGA

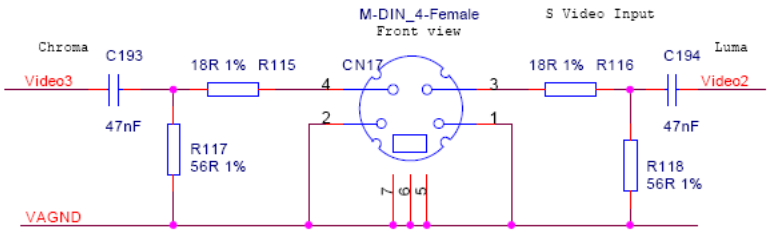


Ilustración 93: RC203-E, entrada S Vídeo

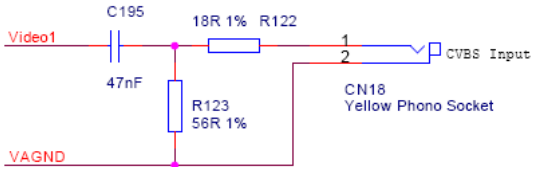


Ilustración 94: RC203-E, conector entrada CVBS

La entrada de vídeo tiene 8 líneas de datos y 6 de control:

Pines entrada Vídeo	Función	Pines FPGA RC203
VIND0 – VIND7	Datos[7:0]	AC22, AC20, AC19, AD19, AC18, AD18, AC17, AC16
VINC0	RTS1	AA22
VINC1	RTS0	R19
VINC2	RTCO	T19
VINC3	SCL	R20

VINC4	SDA	T20
VINC5	CEP	U20

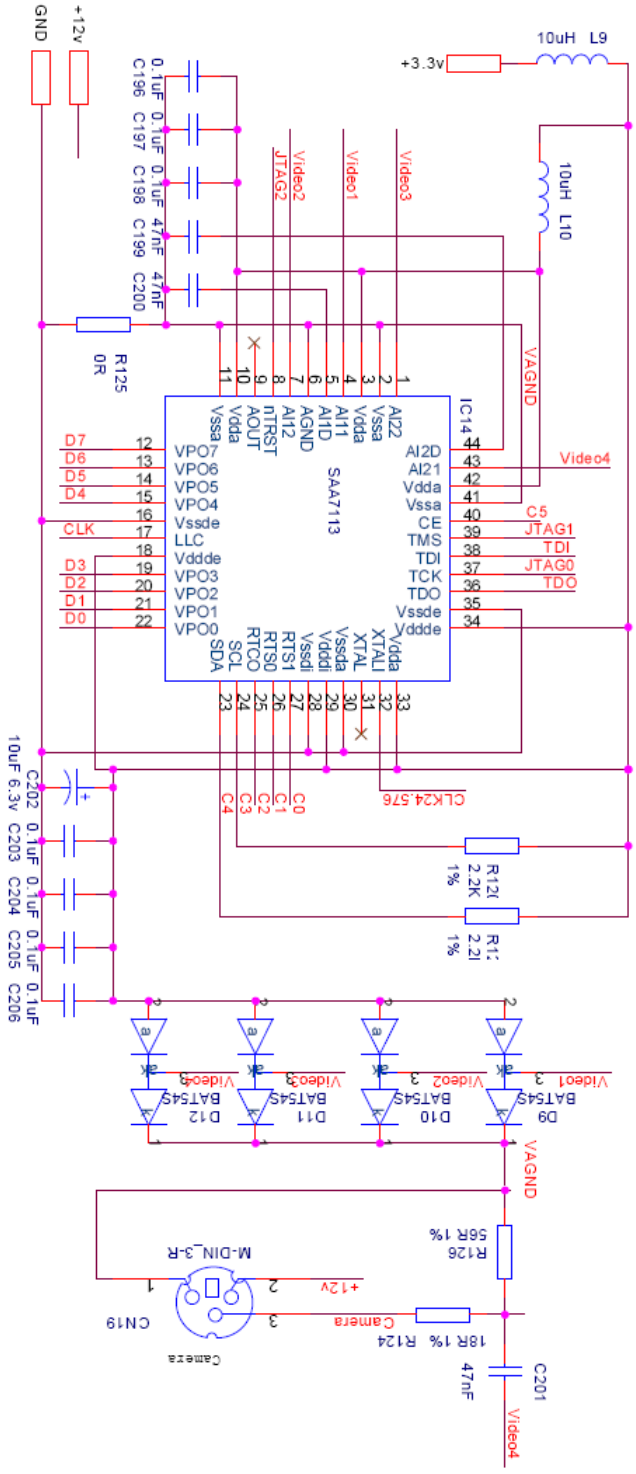


Ilustración 95: RC203-E, cámara de vídeo

4.3.19 Procesado de salida de Vídeo

La tarjeta RC203 puede convertir la entrada digital RGB a salidas para una pantalla VGA, TV (PAL o NTSC) o LCD:

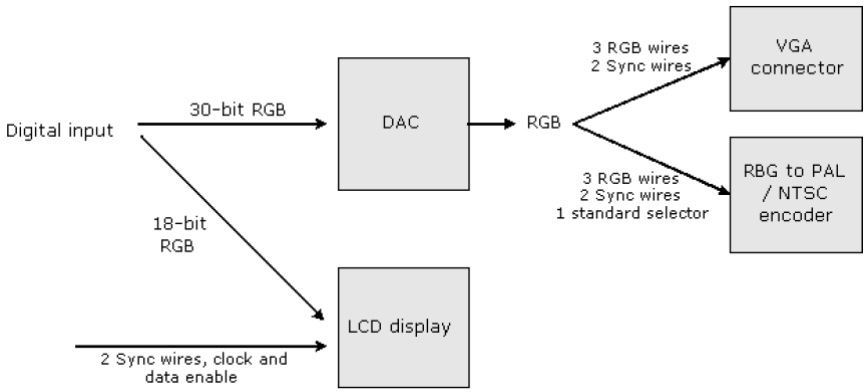


Ilustración 96: RC203-E, procesado de la señal de salida de vídeo

4.3.19.1 DAC

EL CI de Analog Devices ADV7123 DAC de vídeo a alta velocidad puede convertir una entrada de 30 bits a salida VGA o a entrada RGB para el encoder NTSC/PAL.

Pines DAC	Función	Pines de la FPGA
RGB0 - RGB9	Rojo [9:0]	W20, Y18, Y17, Y16, Y15, W16, W15, AD12, AC12, AD11
RGB10 – RGB19	Verde [9:0]	AC11, AC10, W13, Y13, AB13, AB12, AA12, AD20, AD17, AB11
RGB20 – RGB29	Azul [9:0]	AA11, AB10, AA10, AB9, AA9, AB8, AA8, AD10, AD7, W12
RGB30	Pin de reloj	W11
RGB31	Pin Not blank	Y12
RGB32	Pin No Sincronización	Y11
RGB33	Pin Sinc. Vertical	Y10
RGB34	Pin Sinc. Horizontal	Y9
RGB35	Pin monitor SDA	Y8
RGB36	Pin monitor SCL	Y7

4.3.19.2 Encoder RGB a NTSC/PAL

La tarjeta RC203 posee un CI de Analog Devices AD725, es un encoder de RGB a NTSC/PAL. Este recibe una entrada RGB desde el conversor analógico-digital (DAC) de vídeo.

NTSC/PAL encoder pins	Función	Pines FPGA RC203
TV0	Pin estándar	AD16
TV1	Pin sincronización Horizontal	AD15
TV2	Pin sincronización Vertical	AD14

4.3.20 TFT flat panel display

La pantalla TFT (thin film transistor) es un Optrex T-51382D064J-FW-P-AA. El TFT está conectado directamente a la FPGA:

TFT control pins	Función	Pines FPGA RC203
LCD0	Pin Reloj	AC14
LCD1	Pin sincronización horizontal	AA19
LCD2	Pin sincronización Vertical	AB19
LCD3	Pin activación (Data enable)	AA18

El TFT tiene 18 pines de datos: RGB4 - RGB9, RGB14 - RGB19 y RGB24 - RGB29. Estos pines están comparidos entre el TFT y el DAC.

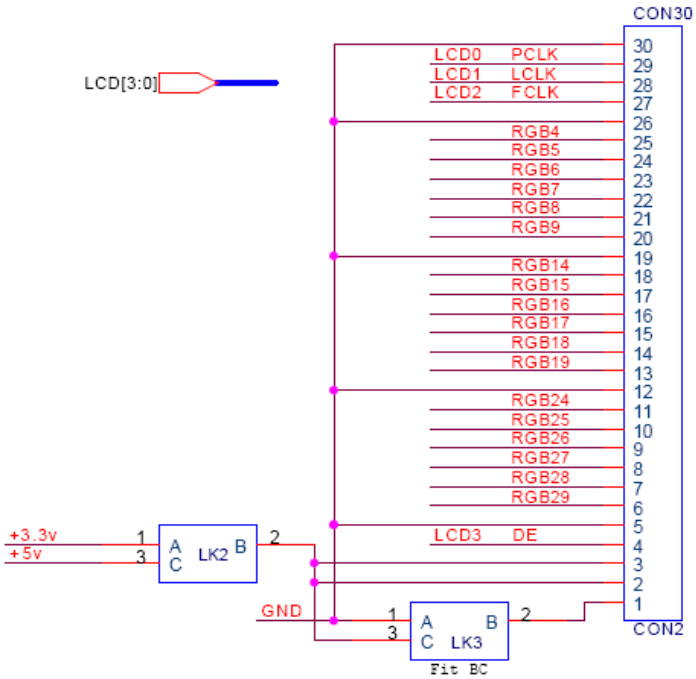


Ilustración 97: RC203-E, conector LCD

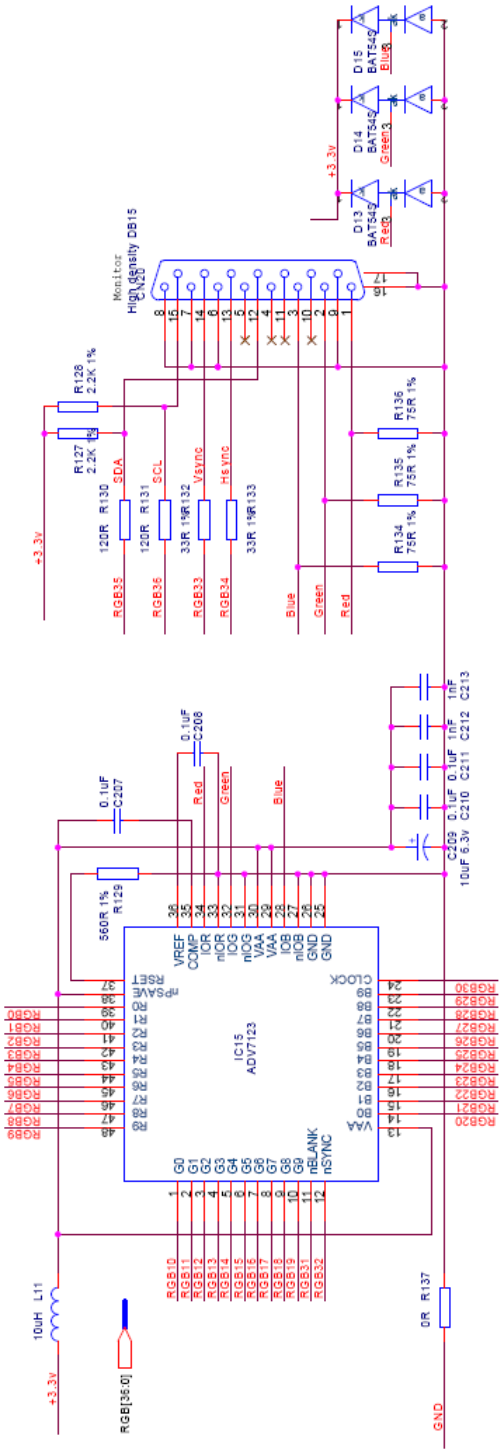


Ilustración 98: RC203-E, conector monitor

5 Diseño de sistemas electrónicos

Este capítulo comienza con una introducción a como afrontar el problema de del diseño de diseño sistemas electrónicos, y el tipo de herramientas que se precisan. Se introducen los conceptos de diseño basados en plataformas electrónicas, así como su abstracción y aplicaciones típicas. Especialmente se estudiará el diseño mediante FPGAs, viendo las etapas típicas de desarrollo e introduciendo conceptos importantes validos para otras plataformas.

Posteriormente, se realizará un examen más detallado a las herramientas proporcionadas por Celoxica, viendo el lenguaje de programación Handel-C.

Este capítulo contiene el manual de programación mediante handel-C, no analizando el entorno de programación, ya que este cometido se verá en capítulos posteriores.

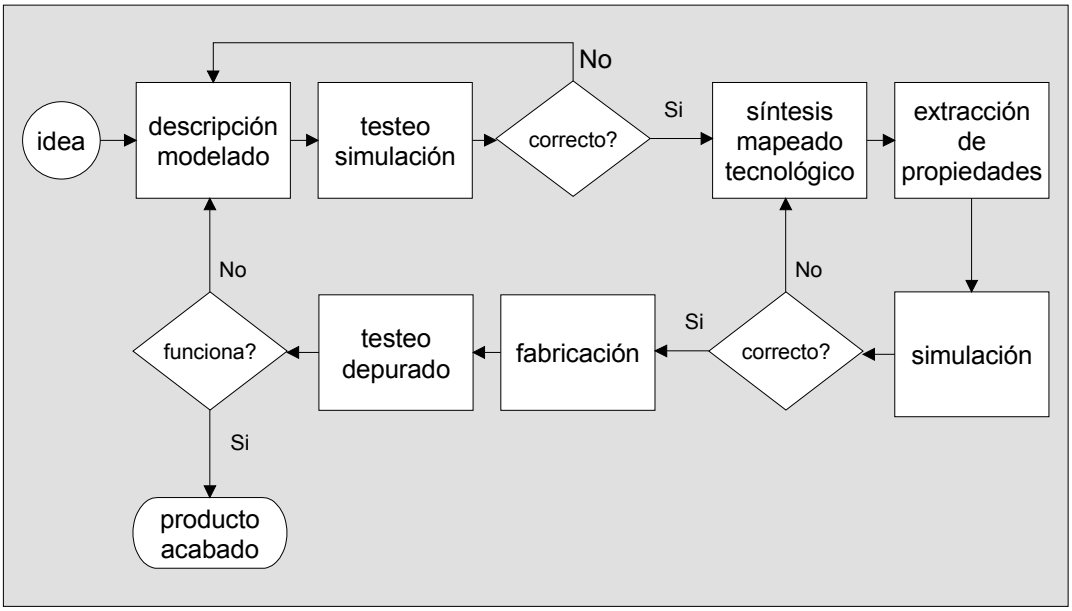
5.1 Herramientas CAD-EDA

CAD acrónimo de Computer Aided Design, es el proceso de diseño que emplea sofisticadas técnicas gráficas de ordenador, apoyadas en paquetes de software para ayudar en los problemas analíticos, de desarrollo, de coste y ergonómicos asociados con el trabajo de diseño.

En principio, el CAD es un término asociado al dibujo como parte principal del proceso de diseño, sin embargo, dado que el diseño incluye otras fases, el término CAD se emplea tanto como para el dibujo, o diseño gráfico, como para el resto de herramientas que ayudan al diseño, como la comprobación de funcionamiento, análisis de costes, etc.

EDA acrónimo de Electronic Design Automation, es el nombre que se le da a todas las herramientas, tanto hardware como software, para la ayuda al diseño de sistemas electrónicos.

5.1.1 Flujo de diseño para sistemas electrónicos y digitales



En el ciclo de diseño hardware las herramientas de CAD están presentes en todos los pasos. En primer lugar en la fase de descripción de la idea, que será un esquema eléctrico, un diagrama de bloques, etc. En segundo lugar en la fase de simulación y comprobación de circuitos, donde diferentes herramientas permiten realizar simulaciones de eventos, funcional, digital o eléctrica de un circuito atendiendo al nivel de simulación requerido.

Por último existen las herramientas de CAD orientadas a la fabricación. En el caso de diseño hardware estas herramientas sirven para la realización de PCBs (Printed Circuit Boards o placas de circuito impreso), y también para la realización de ASICs (Application Specific Integrated Circuits) herramientas que permiten la realización de microchips así como la realización y programación de dispositivos programables.

5.1.2 Herramientas CAD para el diseño hardware

- ***Lenguajes de descripción de circuitos***. Son lenguajes mediante los cuales es posible describir un circuito eléctrico o digital. La descripción puede ser de bloques, donde se muestra la arquitectura del diseño, o de comportamiento, donde se describe el comportamiento del circuito en vez de los elementos de los que está compuesto.
- ***Captura de esquemas***. Es la forma clásica de describir un diseño electrónico y la más extendida ya que era la única usada antes de la aparición de las herramientas de CAD. La descripción está basada en un diagrama donde se muestran los diferentes componentes de un circuito.
- ***Grafos y diagramas de flujo***. Es posible describir un circuito o sistema mediante diagramas de flujo, redes de Petri, máquinas de estados, etc. En este caso será una descripción gráfica pero, al contrario que la captura de esquemas, la descripción sería comportamental en vez de una descripción de componentes.
- ***Simulación de sistemas***. Estas herramientas se usan sobre todo para la simulación de sistemas. Los componentes de la simulación son elementos de alto nivel como discos duros, buses de comunicaciones, etc. Se aplica la teoría de colas para la simulación.
- ***Simulación funcional***. Bajando al nivel de circuitos digitales se puede realizar una simulación funcional. Este tipo de simulación comprueba el funcionamiento de circuitos digitales de forma funcional, es decir, a partir del comportamiento lógico de sus elementos (sin tener en cuenta problemas eléctricos como retrasos, etc.) se genera el comportamiento del circuito frente a unos estímulos dados.
- ***Simulación digital***. Esta simulación, también exclusiva de los circuitos digitales, es como la anterior con la diferencia de que se tienen en cuenta retrasos en la propagación de las señales digitales. Es una simulación muy cercana al comportamiento real del circuito y prácticamente garantiza el funcionamiento correcto del circuito a realizar.
- ***Simulación eléctrica***. Es la simulación de más bajo nivel donde las respuestas se elaboran a nivel del transistor. Sirven tanto para circuitos analógicos como digitales y su respuesta es prácticamente idéntica a la realidad.
- ***Realización de PCBs***. Con estas herramientas es posible realizar el trazado de pistas para la posterior fabricación de una placa de circuito impreso.
- ***Realización de circuitos integrados***. Son herramientas de CAD que sirven para la realización de circuitos integrados. Las capacidades gráficas de estas herramientas permiten la realización de las diferentes máscaras que intervienen en la realización de circuitos integrados.
- ***Realización de dispositivos programables***. Con estas herramientas se facilita la programación de este tipo de dispositivos, desde las más simples PALs (Programmable And Logic) hasta las más complejas FPGAs (Field Programmable Gate Arrays), pasando por las PLDs (Programmable Logic Devices)

5.2 Tipos de Diseños

- Diseño de software
- Diseño de hardware
- Codiseño de Sw/Hw
- Diseño y codiseño con FPGAs

5.2.1 Diseño de Software

Mediante el uso de un conjunto de instrucciones interpretadas por una unidad de ejecución se realiza el procesamiento de datos. Los lenguajes y entornos de programación facilitan la implementación de las aplicaciones generando el conjunto de instrucciones. La tarea principal del programador es implementar los diversos algoritmos que componen la aplicación usando las herramientas de programación.

5.2.2 Diseño de Hardware

Mediante el uso o diseño de un conjunto de componentes con ciertas propiedades eléctricas se establece el circuito de cargas portadoras entre los diferentes componentes y a través de ellos.

Un valor añadido de muchos componentes electrónicos es la posibilidad de realizar diseño lógico programable (por ejemplo las FPGAs). Son sistemas en los que se puede especificar el comportamiento de ciertos dispositivos.

5.2.3 Codiseño de Sw/Hw

A grandes rasgos es el desarrollo de una aplicación realizando tanto el diseño de software como hardware. No obstante, en este grupo también están las implementaciones de algoritmos en dispositivos lógicos programables, donde las FPGAs son el dispositivo más utilizado para realizar este tipo de diseños.

5.2.4 Diseño y codiseño con FPGAs

Cuando se realiza el diseño de una aplicación bajo soporte electrónico, se puede recurrir a un software para una determinada plataforma específica, siendo esta solución barata y flexible, no obstante puede que surja la necesidad de implementar una parte o su totalidad con hardware dedicado, aumentando de esta manera las prestaciones de nuestro diseño.

Si se decide implementar un diseño bajo un hardware dedicado, hay diferentes opciones, pero hay algunos parámetros que se tendrán en cuenta:

- Tiempo de su diseño e implementación.
- Nivel de prestaciones requeridas
- Tiempo en introducir la aplicación al mercado
- Coste estimado de su fabricación

Las FPGA's proporcionan la posibilidad de programar los circuitos in situ, también reducen el ciclo de diseño ya que se pueden sacar prototipos relativamente con poco tiempo, reduciendo además el tiempo para sacar la aplicación al mercado, y añadiendo la posibilidad de la reconfiguración del circuito electrónico. Entre los inconvenientes de su utilización, están su baja velocidad de operación y baja densidad lógica (poca lógica implementable en un solo chip). Su baja velocidad se debe a los retardos introducidos por los conmutadores y las largas pistas de conexión.

5.2.4.1 Aplicaciones típicas

Las características de las FPGA son su flexibilidad, capacidad de procesado en paralelo y velocidad. Esto les convierte en dispositivos idóneos para:

- Simulación y depuración en el diseño de microprocesadores.
- Simulación y depuración en el diseño de ASICs.
- Procesamiento de señal digital, por ejemplo vídeo.
- Sistemas aeronáuticos y militares.

En Internet hay disponible código fuente de sistemas como microprocesadores, microcontroladores, filtros, módulos de comunicaciones, memorias, etc. Estos códigos se llaman cores.

Un soft IP-core (soft intellectual property core) es un modelo sintetizable de una cierta componente HW, del que se puede obtener una implementación física en funcionamiento si se suministra como entrada a una herramienta de diseño HW automático. El objetivo final que se persigue cuando se diseñan sistemas HW usando IP-cores es abaratar costes. En primer lugar reduciendo el ciclo del diseño, ya se que libera al diseñador del desarrollo y prueba de algunas partes del circuito; y en segundo lugar permitiendo el reuso de una misma porción del diseño en diferentes proyectos.

5.2.4.2 Diseño

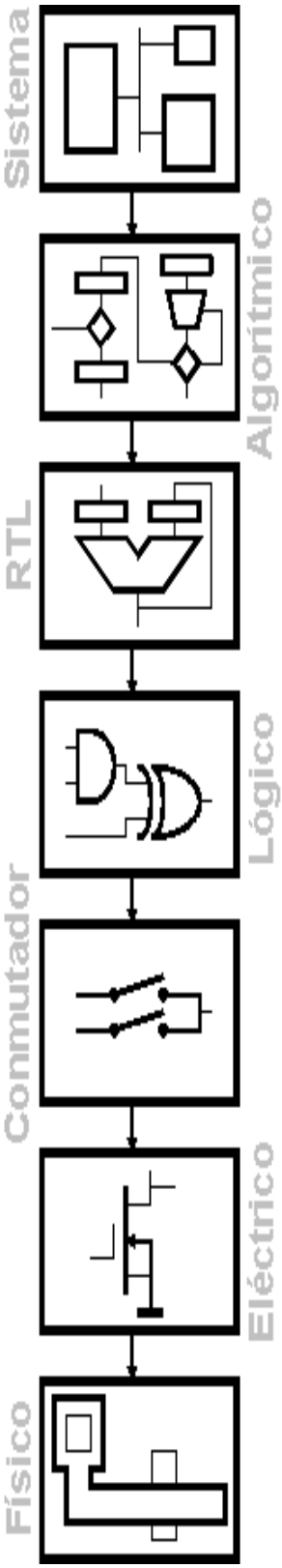
El proceso de diseño de un circuito digital utilizando una matriz lógica programable puede descomponerse en dos etapas básicas:

1. Dividir el circuito en bloques básicos, asignándolos a los bloques configurables del dispositivo.
2. Conectar los bloques de lógica mediante los conmutadores necesarios.

5.2.4.3 Programación

La tarea del programador es definir la función lógica que realizará cada uno de los CLB, seleccionar el modo de trabajo de cada IOB e interconectarlos todos.
El diseñador cuenta con la ayuda de herramientas de programación. Cada fabricante suele tener las suyas, aunque usan unos lenguajes de programación comunes, los HDLs.

5.3 Niveles de Abstracción



Nivel sistema. Describe el sistema como un conjunto de módulos semi-autónomos y cooperantes, cuya interacción se analiza para obtener un conjunto óptimo. No se especifica la forma de realizar cada uno de los módulos.

Nivel algorítmico, funcional o de comportamiento. Cada módulo del nivel superior se define mediante un algoritmo en un lenguaje de alto nivel (HLL). Dada la naturaleza paralela del hardware, en el que varios procesos pueden necesitar un mismo recurso (por ejemplo, un bus de datos) o en el que se deben sincronizar procesos independientes, se utilizan instrucciones muy similares a las de los lenguajes de programación concurrentes, como por ejemplo, semáforos y regiones críticas.

Nivel RTL (Register Transfer Level) o de flujo de datos (Data-Flow). En él se describe el sistema mediante diagramas de transferencias entre registros, tablas de verdad o ecuaciones lógicas. Se le concede más importancia a lo que hace el sistema que a como lo hace. Los elementos básicos de este nivel son registros, memorias, lógica combinacional y buses. Se distingue entre elementos con capacidad de almacenamiento y elementos sin ella.

Nivel lógico. Consiste en la descripción del sistema mediante la interconexión de bloques básicos, como puertas lógicas y biestables. No se realiza una descripción del comportamiento, sino de la estructura del mismo. Si se incluyen además otro tipo de bloques, en general a este nivel se le suele denominar también estructural.

Nivel conmutador. Las puertas se sustituyen por transistores considerados como conmutadores ideales que toman los valores cero o uno. La descripción es, por lo tanto, básicamente la misma que en el nivel anterior con la única diferencia de que en algunas tecnologías, como por ejemplo CMOS, surgen nuevos tipos de circuitos porque los conmutadores pueden ser bidireccionales mientras que las puertas son unidireccionales.

Nivel eléctrico. Se describe el sistema mediante modelos reales del transistor, con sus diferentes parámetros eléctricos.

Nivel físico. Está constituido por la descripción geométrica o simbólica de las máscaras que se emplean para la fabricación el circuito.

5.4 Metodologías según su abstracción

Una metodología describe el modo en el que se realizara el diseño:

- No jerárquica
  - Flat
    - Se describe todo el circuito de una sola vez, en un único bloque.
    - Adecuada para diseños pequeños
- Jerárquica
  - Basadas en la estrategia de "divide y vencerás"
  - Se describe el circuito como un esquema jerárquico de bloques interconectados
  - Adecuada para diseños grandes
  - Dos tipos principales:
    - Botton-Up
    - Top-Down

5.4.1 Metodología Bottom-Up

La metodología de diseño Botton-Up (de abajo hacia arriba) se aplica al método de diseño mediante el cual se realiza la descripción del circuito o sistema que se pretende realizar, empezando por describir los componentes más pequeños del sistemas para, más tarde, agruparlos en diferentes módulos, y estos a su vez en otros módulos hasta llegar a uno solo que representa el sistema completo que se pretende realizar. Esta metodología de diseño no implica una estructuración jerárquica de los elementos del sistema.

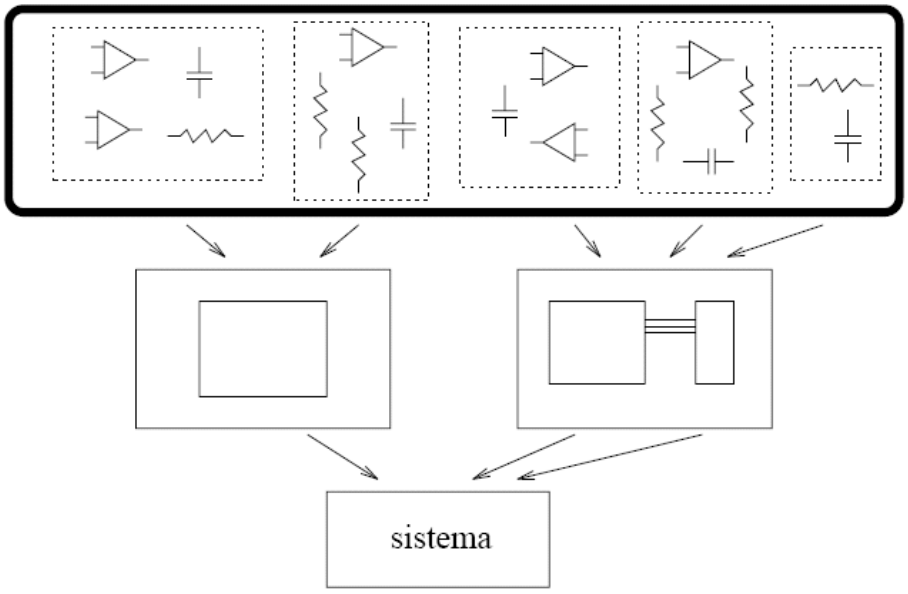


Ilustración 99: Metodología Bottom-Up

5.4.2 Metodología Top-Down

El diseño Top-Down se basa en lema de “divide y vencerás", de manera que un problema, en principio muy complejo, es dividido en varios subproblemas que a su vez pueden ser divididos en otros problemas mucho más sencillos de tratar. En el caso de un circuito esto se traduciría en la división del sistema completo en módulos, cada uno de los cuales con una funcionalidad determinada. A su vez, estos módulos, dependiendo siempre de la complejidad del circuito inicial o de los módulos, se pueden dividir en otros módulos hasta llegar a los componentes básicos del circuito o primitivas.

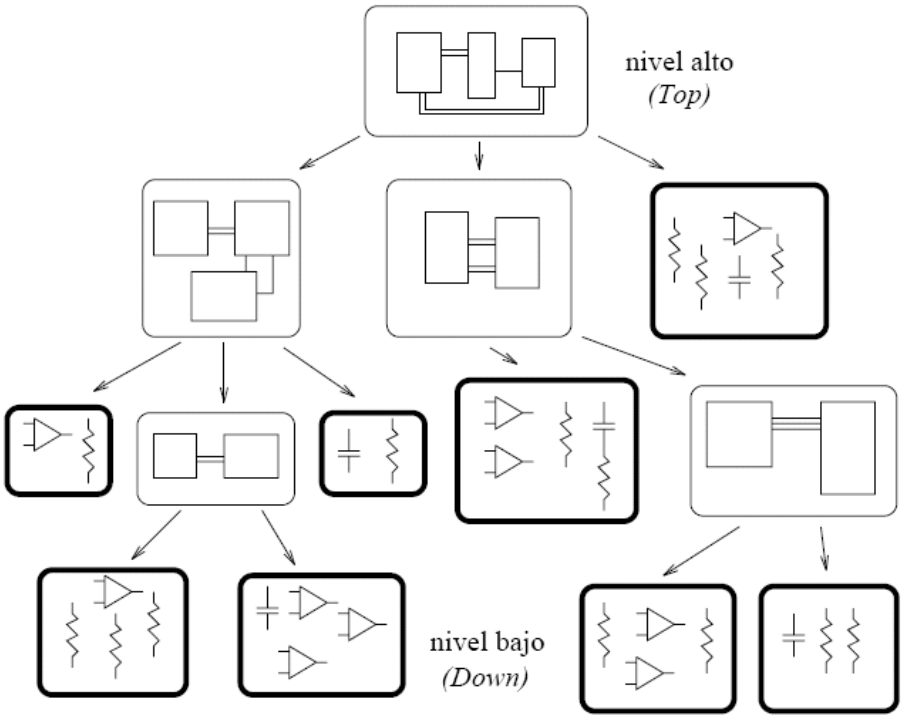


Ilustración 100: Metodología Top-Down

5.4.2.1 Diseño modular

El flujo de diseño top-down, ofrece una ventaja adicional, y es que la información se estructura de forma modular. El hecho de empezar la realización de un diseño a partir del concepto de sistema, hace que las subdivisiones se realicen de forma que los diferentes módulos generados sean disjuntos entre sí y no se solapen. De esta forma, el diseño modular será la realización de diseños realizando divisiones funcionalmente complementarias de los diversos componentes del sistema, permitiendo de esta manera una subdivisión clara y no solapada de las diferentes tareas dentro del diseño.

El diseño bottom-up, no ofrece tanta facilidad para la división del diseño en partes funcionalmente independientes. Al partir de los elementos básicos de los que se compone el sistema, no resulta tan sencillo agruparlos de forma coherente. Esta es otra de las desventajas del flujo de diseño bottom-up, el resultado final puede resultar bastante confuso al no estar modularmente dividido.

5.4.2.2 Diseño Jerárquico

Un complejo diseño electrónico puede necesitar cientos de miles de componentes lógicos para describir correctamente su funcionamiento. Estos diseños necesitan que sean organizados de una forma que sea fácil su comprensión. Una forma de organizar el diseño es la creación de un diseño modular jerárquico.

Una jerarquía consiste en construir un nivel de descripción funcional de diseño debajo de otro de forma que cada nuevo nivel posee una descripción más detallada del sistema. La construcción de diseños jerárquicos es la consecuencia inmediata de aplicar el flujo de diseño top-down.

5.4.3 Descripción del circuito

Existen dos formas de describir un circuito. Por un lado se puede describir un circuito indicando los diferentes componentes que lo forman y su interconexión, de esta manera tenemos especificado un circuito y sabemos como funciona; esta es la forma habitual en que se han venido describiendo circuitos y las herramientas utilizadas para ello han sido las de captura de esquemas y las descripciones netlist.

La segunda forma consiste en describir un circuito indicando lo que hace o cómo funciona, es decir, describiendo su comportamiento. Naturalmente esta forma de describir un circuito es mucho mejor para un diseñador puesto que lo que realmente le interesa es el funcionamiento del circuito más que sus componentes. Por otro lado, al encontrarse lejos de lo que un circuito es realmente, puede plantear algunos problemas a la hora de realizar un circuito a partir de la descripción de su comportamiento.

5.4.3.1 Netlist

El netlist es la primera forma de describir un circuito mediante un lenguaje, y consiste en dar una lista de componentes, sus interconexiones, y las entradas y salidas. No es un lenguaje de alto nivel, por lo que no describe como funciona el circuito, sino que simplemente se limita a describir los componentes que posee y las conexiones entre ellos.

5.4.3.1.1 Formato EDIF

El formato EDIF (Electronic Design Interchange Format) es un estándar industrial para facilitar el intercambio de datos de diseño electrónico entre sistemas EDA (Electronic Design Automation). Este formato de intercambio está diseñado para tener en cuenta cualquier tipo de información eléctrica, incluyendo diseño de esquemas, trazado de pistas (físicas y simbólicas), conectividad, e información de texto, como por ejemplo las propiedades de los objetos de un diseño.

La sintaxis de EDIF es bastante simple y comprensible, sin embargo, no se pretende que sea exactamente un lenguaje de descripción de hardware con el cual los diseñadores puedan definir sus circuitos, aunque hay algunos que lo utilizan directamente como lenguaje de descripción. La filosofía del formato EDIF es más la de un lenguaje de descripción para el intercambio de información entre herramientas de diseño, que un formato para intercambio de información entre diseñadores. En cualquier caso, siempre es posible describir circuitos utilizando este lenguaje.

5.4.3.2 Modelado y Síntesis de Circuitos

Modelado es el desarrollo de un modelo para simulación de un circuito o sistema previamente implementado cuyo comportamiento, por tanto, se conoce. El objetivo del modelado es la simulación.

En el proceso de síntesis, se parte de una especificación de entrada con un determinado nivel de abstracción, y se llega a una implementación más detallada, menos abstracta. Por tanto, la síntesis es una tarea vertical entre niveles de abstracción, del nivel más alto en la jerarquía de diseño se va hacia el más bajo nivel de la jerarquía.

Cuando las simulaciones se realizan en un nivel superior de abstracción, interviniendo el proceso de síntesis para realizar el modelado, se conocen como las **técnicas de modelado a nivel de transacciones (TLM).** En otras palabras, En lugar de realizar el diseño a nivel RTL (Nivel Bajo), se suele implementar el sistema con un lenguaje de nivel más alto que el RTL; Se requiere un proceso de síntesis para pasar el algoritmo, de un lenguaje de nivel alto a nivel bajo, RTL; Al mismo tiempo que se implementa el código del sistema, se simula el sistema con los modelos de funcionamiento sintetizados a RTL del código, siendo esta tarea transparente al programador. De este modo se consigue el modelado de todo el sistema, con el código escrito en un lenguaje de nivel alto.

5.5 HDL, Lenguaje de descripción de hardware

> Un **lenguaje** para máquinas, según la real academia de la lengua española es un *"Conjunto de instrucciones codificadas que una computadora puede interpretar y ejecutar directamente."*. No obstante, cuando aplicamos el término lenguaje a los lenguajes de descripción de hardware la definición de lenguaje, seria *"El conjunto de operaciones codificadas que una máquina puede ejecutar directamente."*
>
> Tomando prestada una de las definiciones de programa que nos proporciona el diccionario de la lengua española de la real academia, un **programa** es *"cada una de las operaciones que, en un orden determinado, ejecutan ciertas máquinas."*
>
> En el transcurso de la lectura de este documento también aparecerá el término **aplicación**, y la real academia de la lengua española nos proporciona en una de sus definiciones, *"programa preparado para una utilización específica…"*.

HDL es el acrónimo de *Hardware Description Language* (Lenguaje de Descripción de Hardware). Son lenguajes de programación en los que el objetivo es programar un circuito electrónico.
El flujo de diseño es:

1. Definir la tarea o tareas que tiene que hacer el circuito.
2. Escribir el programa usando un lenguaje **HDL**.
3. Comprobación de la sintaxis y simulación del programa.
4. Programación del dispositivo y comprobación del funcionamiento.

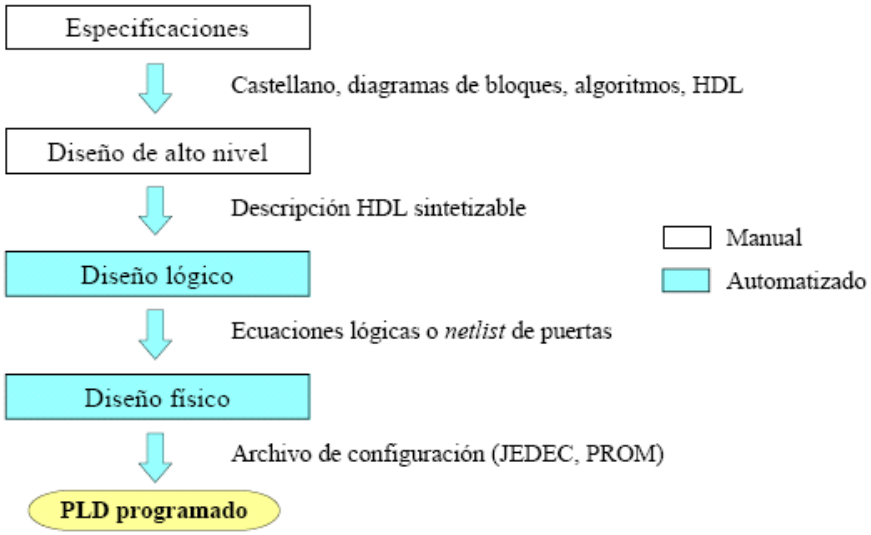


Ilustración 101: HDL, flujo de diseño

Un rasgo común a estos lenguajes suele ser la independencia del hardware y la modularidad o jerarquía, es decir, una vez hecho un diseño éste puede ser usado dentro de otro diseño más complicado y con otro dispositivo compatible.

5.5.1 Tipos de HDLs

Hoy en día existen una gran infinidad de HDLs, no obstante, como ocurre con otros lenguajes de programación, se pueden agrupar en dos grandes grupos, según su nivel de abstracción para desarrollar las aplicaciones y según su grado de síntesis.

- Lenguajes de programación de nivel bajo
  - Síntesis alta (HLS)
    - Verilog
    - VHL
    - ABEL
    - …
- Lenguajes de programación de nivel alto (Diseño ESL)
  - Síntesis baja (LLS)
    - SystemC
    - …
  - Síntesis alta (HLS)
    - Handel-C
    - …

5.5.2 HDLs de nivel bajo

La programación con lenguajes HDL de nivel bajo se realiza a un nivel de programación muy cercano a la circuitería interna del dispositivo, nivel RTL. Por este motivo es necesario saber de antemano su estructura interna. Uno de los lenguajes más representivos es VHDL.

5.5.2.1 VHDL

VHDL Acrónimo de **VHSIC** (Very High Speed Integrated Circuit) **HDL,** es un lenguaje para el modelado y síntesis automática de circuitos. Es un lenguaje basado en ADA. Permite describir la funcionalidad y la organización de sistemas hardware digitales, placas de circuitos y componentes.

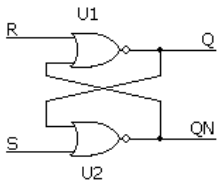
VHDL es un lenguaje que fue diseñado inicialmente para ser usado en el modelado de sistemas digitales. Es por esta razón que su utilización en síntesis no es inmediata, aunque lo cierto es que la sofisticación de las herramientas actuales de síntesis es tal, que permiten implementar diseños especificados en un alto nivel de abstracción. La síntesis a partir de VHDL constituye hoy en día una de las principales aplicaciones del lenguaje con una gran demanda de uso. Las herramientas de síntesis basadas en el lenguaje permiten en la actualidad ganancias importantes en la productividad de diseño.

Algunas ventajas del uso de VHDL para la descripción hardware son:

- VHDL permite diseñar, modelar, y comprobar un sistema desde un alto nivel de abstracción bajando hasta el nivel de definición estructural de puertas.
- Circuitos descritos utilizando VHDL, siguiendo unas guías para síntesis, pueden ser utilizados por herramientas de síntesis para crear implementaciones de diseños a nivel de puertas.
- Al estar basado en un estándar (IEEE Std 1076-1987) los ingenieros de toda la industria de diseño pueden usar este lenguaje para minimizar errores de comunicación y problemas de compatibilidad.
- VHDL permite diseño Top-Down, esto es, permite describir (modelado) el comportamiento de los bloques de alto nivel, analizándolos (simulación), y refinar la funcionalidad de alto nivel requerida antes de llegar a niveles más bajos de abstracción de la implementación del diseño.
- Modularidad: VHDL permite dividir o descomponer un diseño hardware y su descripción VHDL en unidades más pequeñas.

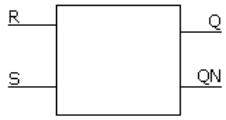
VHDL permite los dos tipos de descripciones:

- **Estructura**: VHDL puede ser usado como un lenguaje de Netlist normal y corriente donde se especifican por un lado los componentes del sistema y por otro sus interconexiones.




```
library IEEE;
use IEE.sttd_logic_1164.all;
entity BasculaRS is
        port ( S, R: in STD_LOGIC;
                Q,QN: buffer STD_LOGIC);
end BasculaRS;
architecture BasculaRS_arch of BasculaRS is
component nor2 port (I0, I1: in STD_LOGIC;
                        O out STD_LOGIC);
end component
begin
        U1: nor2 port map(R,QN,Q);
        U2: nor2 port map(S,Q,QN);
end BasculaRS_arch;
```

- **Comportamiento o funcional**: VHDL también se puede utilizar para la descripción comportamental o funcional de un circuito. Esto es lo que lo distingue de un lenguaje de Netlist. Sin necesidad de conocer la estructura interna de un circuito es posible describirlo explicando su funcionalidad. Esto es especialmente útil en simulación ya que permite simular un sistema sin conocer su estructura interna, pero este tipo de descripción se está volviendo cada día más importante porque las actuales herramientas de síntesis permiten la creación automática de circuitos a partir de una descripción de su funcionamiento.




```
library IEEE;
use IEE.sttd_logic_1164.all;

entity BasculaRS is
        port ( S, R: in STD_LOGIC;
                Q,QN: buffer STD_LOGIC);
end BasculaRS;

architecture BasculaRS_arch of BasculaRS is
begin
        QN <= S nor Q;
        Q <= R nor QN;
end BasculaRS_arch;
```

5.5.3 HDLs de nivel alto

5.5.3.1 Diseño ESL (Electronic System Level)

Los sistemas que hasta hace unos años consistían de decenas de circuitos interconectados en una placa de circuito impreso se puedan implementar actualmente en un único circuito integrado, que se suele denominar sistema en chip (SoC).

Mediante el diseño ESL se diseñan y verifican sistemas (SoC) usando modelos abstractos, concentrando el esfuerzo en la arquitectura del sistema y en los algoritmos, más que en puestas en práctica de sus diseños en nivel bajo RTL. El diseñador puede crear y analizar sistemas complejos a partir de librerías de componentes que han sido descritos a un nivel de abstracción superior utilizando lenguajes de nivel alto.

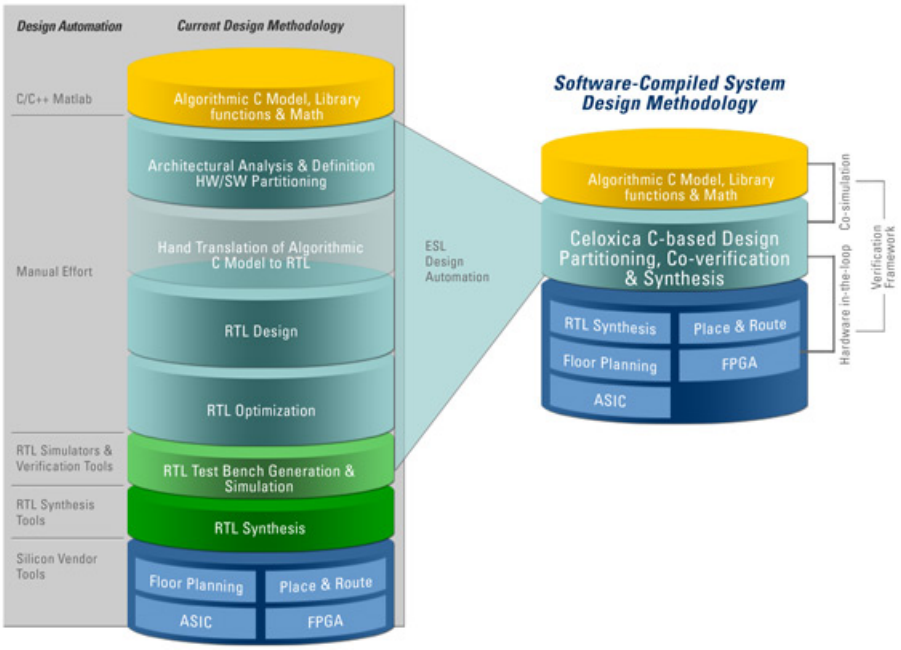


Ilustración 102: Diseño no ESL vs. Diseño ESL de Celoxica

El diseño mediante ESL ofrece soluciones superiores, diseños más rentables con tiempos más cortos de diseño usando técnicas como:

- Para el diseño de una aplicación compleja se especifica bajo un algoritmo en lenguajes de nivel alto (*HLL síntesis*: basados en C Handel-C, basados en C++ SystemC).
- A partir de la exploración de la arquitectura de la FPGA se realiza un mejor mapeado de la aplicación en el área de la FPGA.
- Selección y calificación del IP para la reutilización de bloques preexistentes críticos.
- Especificación de la plataforma como modelo de referencia para el desarrollo.

5.5.3.1.1 Flujo de diseño basado en tres Niveles

Un diseño típico basado en ESL tiene tres niveles. En el nivel más alto está el acercamiento algorítmico. Después del diseño y la implementación el siguiente nivel hace una exploración arquitectónica. Y en el tercer nivel está la fase de generación automática.

5.5.3.1.2 Síntesis por medio de lenguajes de nivel alto (HLL)

La mayoría de los lenguajes utilizados en diseños ESL, están basados en C y el C++, los más representativos son Handel-C y SystemC.

5.5.3.1.3 System Level APIs

Dentro del contexto de diseño ESL una API (la interfaz de programación de aplicación) es una capa que permite a desarrolladores trabajar a un nivel más alto de abstracción de diseño. Esto elimina las distracciones del desarrollador y quita problemas paralelos, como son el soporte de desarrollo y la independencia de hardware prevista por el uso de microprocesadores modernos manejando sistemas operativos.

5.6 Handel-C

Diseñado por Celoxica, es un lenguaje para implementar algoritmos en hardware directamente de una representación en C. Para diseñar sistemas usando Handel-C, se utiliza la herramienta de desarrollo y síntesis digital DK Design Suite. Para realizar el codiseño se utiliza Nexos PDK.

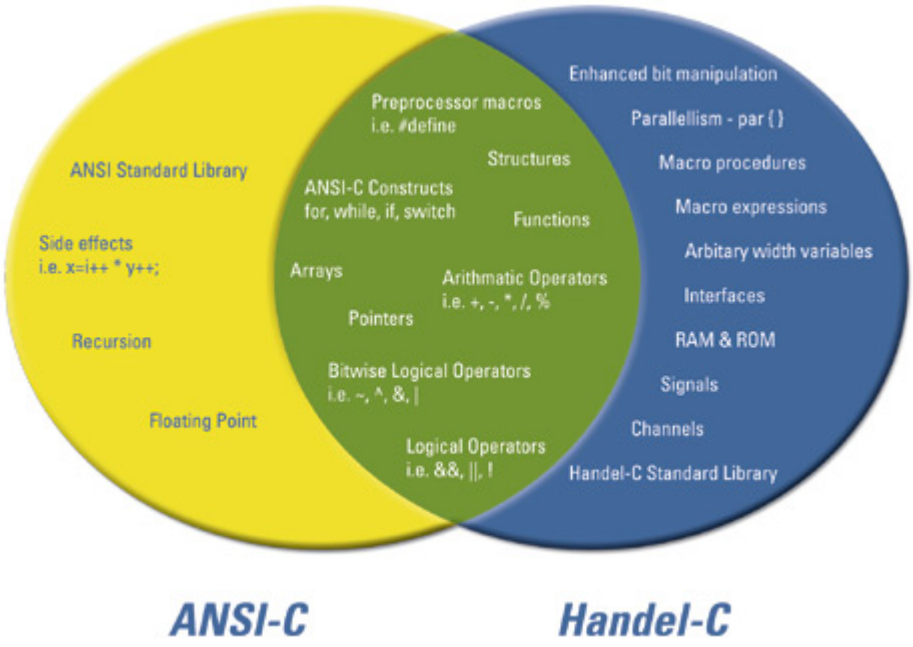


Ilustración 103: Lenguaje programación C: Handel-C y ANSIC

5.6.1 Nociones básicas

Handel-C utiliza la sintaxis de C con la adición de paralelismo inherente. Se pueden escribir programas secuenciales en Handel-C, pero para aprovechar al máximo el hardware, se recomienda realizar construcciones paralelas.

- Programas en Handel-C
- Programas paralelos.
- Comunicaciones por medio de canales.
- Variables:

- Uso compartido y alcance
- Ancho y valor.

- Estructura general y reloj

5.6.1.1 Programas en Handel-C

Un programa escrito en Handel-C es implícitamente secuencial. Escribir una sentencia después de otra, señala que estas instrucciones deberán ser ejecutadas en ese orden exacto.

Para ejecutar instrucciones en paralelo, se debe usar utilizar la palabra reservada. Handel-C provee construcciones para controlar el flujo de un programa. Por ejemplo, el código puede ser ejecutado condicionalmente a merced del valor de alguna expresión, o un bloque de código puede estar repetido un cierto número de veces.

La implementación de un algoritmo en Handel-C es transparente al programador. Es decir, el programador no tiene que preocuparse de las capas inferiores, ya que esta tarea está reservada al compilador. Esta filosofía hace que Handel-C sea un lenguaje de programación en vez de un idioma de descripción del hardware. En algunos sentidos, Handel-C es para el hardware, lo que un idioma de alto nivel convencional es para el lenguaje ensamblador de un microprocesador.

El diseño de hardware realizado con la herramienta DK se basa en el código fuente implementado con el lenguaje de programación Handel-C.

Las puertas lógicas que hacen el circuito final son las instrucciones ensambladas bajo Handel-C.

5.6.1.2 Programas paralelos

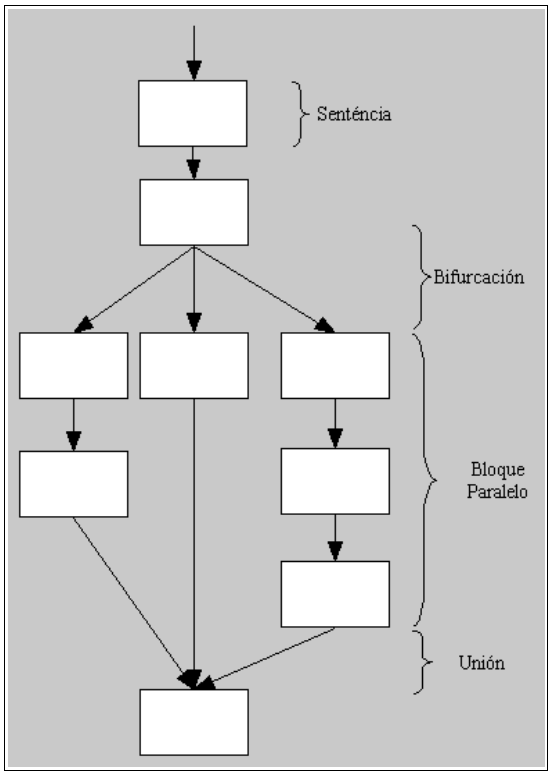


Ilustración 104: Ejemplo de ejecución en paralelo

El paralelismo en Handel-C es paralelismo hardware, realmente se ejecutan las instrucciones en paralelo sin ninguna multiplexación temporal.

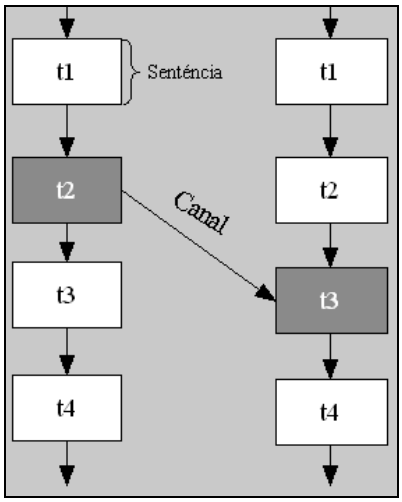
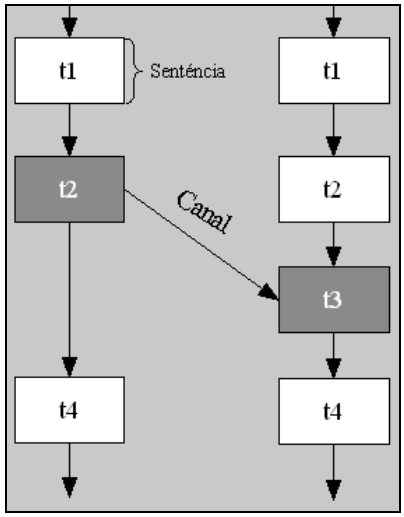
Las instrucciones escritas en paralelo se ejecutaran al mismo tiempo en zonas de hardware distintas.

Cuándo se implementa un bloque paralelo, se crean dos o más hilos de ejecución (ramas) que se ramificaran con las instrucciones de cada rama. Al final de la ejecución en paralelo, las ramificaciones vuelven a unirse en un mismo punto de ejecución. Si una rama de ejecución termina antes que las otras, esta se esperará a que termine el hilo de ejecución (rama) más lento, para continuar.

5.6.1.3 Comunicación por medio de canales

En una ejecución en paralelo, los canales proporcionan un enlace entre las ramas. Una rama coloca la información en un canal y la otra la lee.

Los canales pueden construirse con y sin aptitudes de salida en orden de entrada (FIFO):

No bloqueantes (con FIFO)	**Bloqueantes (sin FIFO):**
	
El emisor coloca los datos en el canal y continúa su ejecución. No obstante el receptor para su ejecución hasta que no haya datos en el canal.	El emisor coloca los datos en el canal y para su ejecución hasta que no hay una petición por parte del receptor. Una ver se ha atendido con éxito la petición de datos ambos canales continúan su ejecución. Esta comunicación sirve para sincronizar canales.

5.6.1.4 Variables

5.6.1.4.1 Uso compartido y alcance

El alcance de declaraciones gira alrededor de bloques de código. Un bloque de código es delimitado mediante corchetes, { ... }. Esto significa que:

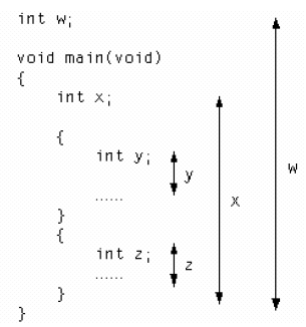


Ilustración 105: Alcance de variables

- Las variables globales deben ser declaradas fuera de todos los bloques de código.
- Una variable está en alcance dentro de un bloque de código y cualquiera de los sub-bloques de ese bloque.

5.6.1.4.2 Ancho y valor

Una diferencia crucial entre Handel-C y C, es la capacidad de Handel-C para trabajar con variables de ancho de bits arbitrario. Normalmente C opera con variables de anchos de bits de 8, 16, 32 y 64bits, debido al ancho de palabra de los procesadores actuales. Es relativamente complicado trabajar con otras anchuras. Handel-C puede trabajar con variables de cualquier ancho de bits. Handel-C, además permite unir varias variables para implementar anchos mayores.

5.6.1.5 Estructura general del programa

La estructura general de un programa en Handel-C difiere a la estructura general de un programa escrito en ANSI-C:

```
Declaraciones Globales

Definición del reloj

void main(void)
{
        Declaraciones Locales

        Código
}
```

FIG: Estructura general de un programa en Handel-C

- Una o más funciones principales “main”.
- Cada una de las funciones “main” tiene que tener asociado un reloj.
- Las funciones “main” no tienen parámetros ni devuelven valor alguno.

De este modo se pueden implementar diferentes módulos con diferentes frecuencias de reloj.

```
set clock = external;                                   // Configura el reloj
void main(){                                            // Inicio del diseño
        static unsigned 32 a = 238888872, b = 12910669; // variables de entrada
        unsigned 32 Result;                             // Variable de salida
        interface bus_out() OutputResult(Result);       // Result sale por los
                                                        // pines
        while (a != b){
                if (a > b)
                        a = a - b;
                else
                        b = b - a;
        }
        Result = a;                                     // Carga la variable de
}                                                       // salida
```

FIG: Ejemplo programa en Handel-C, 1 función main con Reloj externo

5.6.2 Fundamentos del lenguaje Handel-C

- Handel-C es un lenguaje tipo C:
  - Sintaxis y semántica ANSI-C.
  - Extensiones y restricciones para el diseño hardware.
- Diseñado para el diseño hardware síncrono:
  - Optimizado para FPGAs.
  - Todo lo que se simula, compila en hardware.
- Extensiones a C permiten producir hardware eficiente:
  - Par introduce el paralelismo.
  - Anchura arbitraria de palabra.
  - Sincronización.
  - Interfaces hardware.

5.6.2.1 Comentarios

Handel-C utiliza los delimitadores estándares para delimitar los comentarios. Los comentarios no deben ser encadenados.

```
/* Comentario valido*/
/* Este NO /* es un */ comentario válido */
```

Handel-C también proporciona comentarios tipo C++. La marca // especifica al compilador que desde la marca // hasta el final de la línea será un comentario.

```
x = x + 1; // Esto es un comentario
```

5.6.2.2 Variables

5.6.2.2.1 Tipos lógicos

Los tipos lógicos representan enteros y caracteres. Cuando se habla de longitud o tamaño, se especifica el número de bits que se utilizan para declarar un tipo lógico. Todos los tipos lógicos tienen asociados un determinado tamaño, con la excepción del tipo lógico entero (int), en el cual, puede especificarse el tamaño.

Tipo	Tamaño en bits
int	*(1)*
[signed | unsigned] int n	n
[signed | unsigned] int	*(2)*
[signed | unsigned] char	8
[signed | unsigned] short	16
[signed | unsigned] long	32
[signed | unsigned] int32	32
[signed | unsigned] int64	64

(1) El tamaño será establecido por el compilador a menos que se utilice el comando '`set intsize = n`'.
(2) El compilador indicará como no definido.

5.6.2.2.1.1 Declaración de enteros

Por omisión los enteros son signed, es decir pueden ser positivos y negativos, las siguientes declaraciones son idénticas:

```
signed int 5 x;
int 5 x;
signed 5 x;
```

Para declarar un número entero positivo utilizaremos la palabra reservada unsigned, seguida de su longitud. También se puede declarar un entero positivo utilizando la macro log2ceil para calcular su longitud. Log2ceil es una macro de la librería estándar stdlib.hch. Las tres declaraciones son idénticas:

```
unsigned (log2ceil(5768)) w;        // 13 bits de longitud
unsigned int 13 w;                  // 13 bits de longitud
unsigned 13 w;                      // 13 bits de longitud
```

Se puede utilizar la palabra reservada del compilador "undefined" para omitir la declaración de la longitud de la variable, el compilador automáticamente establecerá el tamaño correcto de esta.

```
set clock = external "p1";
set intTamaño = 27;
void main(void)
{       unsigned x;                 // longitud 27
        unsigned undefined y;       // longitud 5
        unsigned 5 z;               // longitud 5

        z=100;
        y=z;
}
```

5.6.2.2.1.2 Dominio y rango de los enteros

Los enteros declarados como signed tiene representación en complemento a dos, por lo que puede representar tanto enteros positivos como negativos con una única representación para el 0, su rango es de $[-2^{n-1}, +2^{n-1}-1]$.

Para representar un número positivo en complemento a dos, se representa igual que en signo y magnitud, es decir, el bit de mayor peso será 0 seguido de la representación en binario de dicho número. Para representar un número negativo en complemento a dos, primero se invierte el numero en representación binaria estándar (los 0's se convierten a 1's, y los 1's a 0's) y a continuación se le suma 1 al numero invertido. Los números negativos en complemento a dos el bit de más peso es siempre 1, indicando de este modo su signo, es decir negativo.

En cambio los enteros declarados como unsigned, es decir positivos, su rango es: $[0, 2^{n+1}-1]$ o lo que es lo mismo $[0, 2^n)$.

5.6.2.2.1.3 Conversión de enteros

Para hacer compatibles variables del mismo tipo con distintos anchos:

- Sin signo (unsigned):

```
variable_sin_signo_destino = adju(variable_sin_signo_origen, ancho_nuevo)
```

- Con signo (signed):

```
variable_con_signo_destino = adjs(variable_con_signo_origen, ancho_nuevo)
```

Otras conversiones, entre signos y/o tamaños:

- unsigned W1 → unsigned W2

```
unsigned int 8 w;          // W2 = 8, sin conversión
unsigned int 6 x;          // W2 = 6, puede haber perdida de bits
unsigned int 9 y;          // W2 = 9
unsigned int 8 z;          // W1 = 8

z = 0xFF;

w = z;                     // w = 0xFF=255, W1 = W2
y = 0@z;                   // y =  0xFF=255, W1 < W2
y = adju(w, width(z));     // y =  0xFF=255, para todo W1, W2
x = adju(w, width(z));     // x =  0x3F=63, hay perdida de bits
```

NOTA: si W1>W2 puede haber perdida de bits

- unsigned W1 → signed W2

```
signed int 7 w;            // W2 = 6, puede haber perdida de bits
signed int 8 x;            // W2 = 7, puede haber perdida de signo
signed int 9 y;            // W2 = 8
unsigned 8 z;              // W1 = 8

z = 0xFF;                  // z = 0xFF=255;

w = (int 7) adju(x,width(w));  // w= -1= 0x7F, Hay perdida de bits
x = (int 8) adju(x,width(x));  // x=  -1=  0xFF, hay perdida de signo
y = (int 9) adju(x,width(y));  // y= 255=0x0FF
```

NOTA: si W1<W2-1, hay perdida de valor (bits y signo)
W1<W2-1, es igual a decir, W1≤W2

- signed W1 → signed W2, W1 ≠ W2

```
int 8 w;                   // W1 = 8
int 16 x;                  // W2 = 16

w = -5;
x = adjs(w, width(x));     // x = -5, para todo W1, W2
```

NOTA: si W1>W2, hay perdida de valor (bits y signo)

- signed W1 → unsigned W2

```
unsigned 8 w;              // W1 = 8
int 8 x;                   // W2 = 9
int 6 z;                   // W2 = 6

w = -5;
z = adjs(w,width(z));      // z = -5
x = (unsigned 8) abs(w);   // x = 5, si W1 = W2
```

```
x = (unsigned 8) ( 0@abs(w) );          // x = 5, si W1 < W2
x = adju(abs(z), width(x));             // x = 5, para todo W1, W2
```

NOTA: si W1≥W2, hay perdida de bits

5.6.2.2.2 Constantes

Las constantes pueden ser usadas en expresiones. Las constantes decimales se escriben con simplemente el número decimal, las constantes hexadecimales llevan el prefijo "0x" o "0X", las constantes octales llevan el prefijo "0" y las constantes binarias llevan el prefijo 0b o 0B. Por ejemplo:

```
w = 1234;                   /* Decimal */
x = 0x1234;                 /* Hexadecimal */
y = 01234;                  /* Octal */
x = 0b00100110;             /* Binario */
```

5.6.2.2.3 Enumeraciones

enum especifica una lista de valores constantes numéricas de tipo entero cuyos valores son consecutivos.

```
enum weekdays {MON, TUES, WED, THURS, FRI};
/*MON=0,TUES=1,WED=2,THURS=3,FRI=4*/

/*Especificando el el valor*/
enum weekdays {MON = 9, TUES, WED, THURS, FRI};
/*MON=9,TUES=10,WED=11,THURS=12,FRI=13*/

/*Especificando el tamaño*/
enum weekdays {MON = (unsigned 4)9, TUES, WED, THURS, FRI};
```

Ejemplo:

```
set clock = external "P1";
typedef enum { A, B, C = 43, D } En;

void main(void) {
        En num;
        int undefined result;
        num = (int 7)D;
        result = num;
}
```

5.6.2.2.4 Arrays

En Handel-C se declaran conjuntos de variables de la misma forma que en C. Por ejemplo:

```
int 6 x[7];
```

Declara 7 registros, cada uno de ellos con 6 bits de anchura. Acceder a las variables es exactamente igual que en C. por ejemplo, para acceder la quinta variable del array:

```
x[4] = 1;
```

La primera variable tiene un índice 0, y lo última tiene un índice n-1, donde la n es el número total de variables en el array. Si se utiliza una variable para recorrer el array como índice, la variable debe ser declarada unsigned.

Ejemplo: Inicializa los elementos de un array con el valor del índice.

```
set clock = external;
void main (void)
{           int 6 ax[7];
            unsigned index;

            index=0;
            do{
                        ax[index]= ( 0 @ index );
                        index++;
            }while (index <= 6);
}
```

NOTA: La anchura de índice tiene que ser ajustada en la asignación. Esto es porque el ancho del índice es 3. Se necesitan tres bits para representar un índice que vaya hasta la posición 6, ya que el array tiene 7 elementos, y la primera posición es 0. la operación 0@variable pone los bits de mayor peso a cero cuando para ajustar dos variables de ancho de bits distintos.

5.6.2.2.5 Estructuras

Una estructura (struct), permite agrupar un conjunto de variables, que sirven para crear un nuevo tipo. Una estructura se utilizará para crear varias instancias:

```
struct persona{
            unsigned 8 edad;
            unsigned 1 sexo;
            char trabajo[25];
};                                          // define el tipo persona

strcut persona hermana;                     // creación de una instancia bacía
hermana.edad=30;                            // asignación en un elemento de la estructura

// creación de una instancia con inicialización de variables
static struct Robert = { 25, 1, {‘e’,’t ’,’n ’,’a ’,’i ’,’d ’,’u ’,’t ’,’s ’,’E’}};
```

5.6.2.2.6 Apuntadores

Para declarar un apuntador, se debe especificar el tipo de variable a la que hará referencia el apuntador, y posteriormente se especificará el nombre de la variable antecedida de un * (asterisco).

```
Tipo *Nombre;
```

Los punteros son utilizados para apuntar a variables, para que un puntero haga referencia a la variable se utiliza el operador &. El tipo de variable ha de ser idéntico para el apuntador y la variable de referencia:

```
int 8 *ptr;
int 8 object, x;

object = 6;
x = 10;
ptr = &object;        // ptr se le asigna la direcció de object
x = *ptr;             // x pasa a ser 10
*ptr = 12;            // object pasa a ser 12
```

Los punteros también pueden hacer referencia a funciones y a interfaces:

```
int 8 *  nombre_funcion();                  // función que devuelve un puntero
int 8 (* nombre_puntero)();                 // puntero a una función

 interface bus_out() * p(int 2 x);
interface bus_out() b(int x 2 x=y);         // definición del interface
p=&b;  // p ahora apunta a b
```

Los punteros a estructuras son idénticos al C convencional, para hacer referencia des de un puntero a un miembro de la estructura se utiliza el operador ->

```
struct S{
    int 18 a,b;
} s, *sp;

sp = &s;
s.a = 26;
sp->b = sp->a;
```

Punteros dobles:

```
struct S{
    int 18 a,b;
} s1, s2, *sp, **spp;

sp=&s1;
spp=&sp;
s2=**sp;              //s2=s1
```

5.6.2.3 Señales

- Una señal se comporta como un cable.
  - Toma el valor asignado sólo durante el ciclo de reloj en curso.
- El valor por defecto es indefinido, para asignarle un valor por defecto se declara como estática:
  - Toma ese valor cuando no se le asigna algo en un ciclo de reloj.
  - Una señal estática sin valor inicial explícito se pone por defecto a 0.

```
signal unsigned 8 SignalA;
static signal unsigned 8 SignalB = 5;
static unsigned 8 VarX = 1, VarY = 2;

par{
        SignalA = VarX * 2;  // asigna un valor a SignalA
        VarX = SignalA;              // usa el valor de SignalA
        VarY = SignalA + 1;  // idem
}                                    // VarX = 2, VarY = 3

VarX = SignalB;              // usar el valo por de fecto de SignalB
                                     // VarX = 5
VarY = SignalA;              // No funciona
                                     // el valor de VarY es indefinido
```

- La asignación se evalúa antes de leer en un ciclo de reloj.
- No puede haber dependencias circulares.

```
signal unsigned 8 a, b;

par{
        a = b;
        b = a;
}
```

- Se podría obtener un comportamiento diferente en simulación y en hardware.
- Se puede usar signal sobre una declaración de un array para crear un array de señales

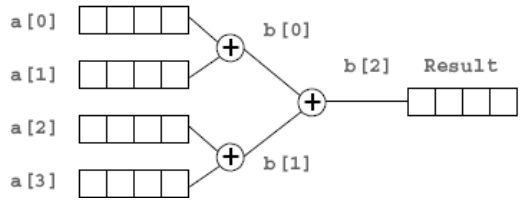


Ilustración 106: Sumador en forma de árbol, mediante señales

```
unsigned 4 a[4];
signal unsigned 4 b[3];
unsigned 4 Result;

while(1){
        par{
                b[0] = a [0] + a[1];
                b[1] = a [2] + a[3];
                b[2] = b [0] + b[1];
                Result = b[2];
        }
}
```

Fig: Sumador en forma de árbol, mediante señales

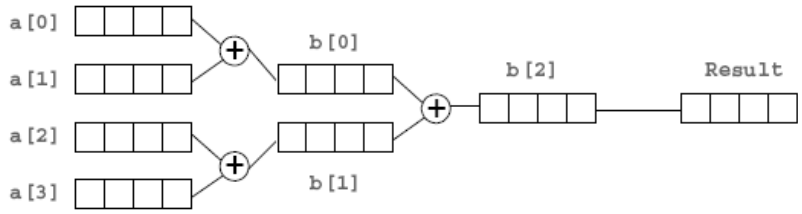


Ilustración 107: Sumador en forma de árbol, mediante registros

```
unsigned 4 a[4];
unsigned 4 b[3];
unsigned 4 Result;

while(1){
        par{
                b[0] = a [0] + a[1];
                b[1] = a [2] + a[3];
                b[2] = b [0] + b[1];
                Result = b[2];
        }
}
```

Fig: Sumador en forma de árbol, mediante registros

5.6.2.4 Operadores

5.6.2.4.1 Orientados a bits

Operador	Significado
<<	Desplazamiento hacia la izquierda
>>	Desplazamiento hacia la derecha
[]	Selección de uno o varios bits
<-	Toma los bits menos significativos
\\	Descarta los bits menos significativos
@	Concatenación
width(expresión)	Devuelve el ancho de una expresión

Desplazamiento a la derecha:

```
set clock = external;
void main (void)
{       unsigned int 4 w;
        unsigned int 4 x;
        signed int 4 y;
        signed int 4 z;

        x = -3;
        y = 3;
        z=2;

        w = 10;             // w = 0b1010 = 10
        x = w >> 1 ;        // x = 0b0101 = 5
        x = w >> 2 ;        // x = 0b0010 = 2

        y = 7;              // y = 0b0111 = 7
```

```
        z =  y >> 1 ;       // z = 0b0011 = 3
        z =  y >> 2 ;       // z = 0b0001 = 1

        y = -8;             // y = 0b1000 = -8
        z = y >> 1 ;        // z = 0b1100 = -4
        z = y >> 2 ;        // z = 0b1110 = -2
        z = y >> 3 ;        // z = 0b1110 = -1
}
```

NOTA: cuando se realiza n desplazamientos a la derecha, se añaden por la izquierda n bits:

- Si se trata de un unsigned, se añaden 0's
- Si se trata de un signed, se añade el bit de mayor peso, el signo.

Desplazamiento a la izquierda:

```
set clock = external;
void main (void)
{       unsigned int 4 w;
        unsigned int 4 x;
        signed int 4 y;
        signed int 4 z;

        w = 10;             // w = 0b1010 = 10
        x = w << 1 ;        // x = 0b0100 = 4
        x = w << 2 ;        // x = 0b1000 = 8

        y = 7;              // y = 0b0111 = 7
        z =  y << 1 ;       // z = 0b1110 = -2
        z =  y << 2 ;       // z = 0b1100 = -4
        z =  y << 3 ;       // z = 0b1000 = -8

        y = -1;             // y = 0b1111= -1
        z = y << 1 ;        // z = 0b1110 = -2
        z = y << 2 ;        // z = 0b1100 = -4
        z = y << 3 ;        // z = 0b1000 = -8
}
```

NOTA: siempre se añaden ceros a la derecha.

Selección de uno o varios bits:

```
set clock = external;
void main (void)
{       unsigned int 8 w;
        unsigned int 1 x;
        unsigned int 5 y;

        w = 0b10001000;     // w = 0b10001000 = 0x88 = 136

        x = w[7] ;          // x = 0b1 = 1
        y = w[7:3] ;        // y = 0b10001 =  0x11 = 17
}
```

Nota:

MSB	Bit más significativo	N-1
LSB	Bit menos significativo	0

Selección y descarte de los bits más significativos:

```
set clock = external;
void main (void)
{       unsigned int 8 x;
        unsigned int 4 y;
```

```
        unsigned int 4 z;
        x = 0xC7;
        y = x <- 4;             // y = 0x7
        z = x // 4;             // z = 0xC
}
```

Concatenación:

```
set clock = external;
void main (void)
{       unsigned int 8 x;
        unsigned int 4 y,z;
        signed int 7 i;
        signed int 12 j;

        y = 0xC;                        // y = 0xC = 12
        z = 0x7;                        // z = 0x7 = 7
        i = -10;

        x = y @ z;                      // x = 0xC7 = 199

        x = (0@y) * (0@z);              // x = 0x54 = 84
                                        // 84 = 17 * 7 ; 0x54 = 0x0C * 0x07;
                                        // los terminos de un producto han de tener el mismo
                                        // ancho, es este caso, 0@ añade 4 ceros a ambos
                                        // operadores

        j =  adjs( i, width(j) );       // para extender el ancho y propagar el signo
        j = i[7] @ i[7] @ i[7] @ i[7] @ i[7] @ i;        // igual que el anterior
}
```

Ancho de una expresión:

```
set clock = external;
void main (void)
{       unsigned int 8 x;
        unsigned int 4 y,z;

        x = 0xC7;

        y = x <- width(y);              // y = 0x7
        z = x // width(z);              // z = 0xC
}
```

Nota: with devuelve el valor del ancho de una expresión, la función width puede ser substituida por una constante, en este ejemplo sería 4.

5.6.2.4.2 Aritméticos

Operador	Significado
+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo

En una operación:

1. Los tipos de las variables tienen que ser idénticos.
2. El ancho de los operandos y el resultado tienen que ser el mismo.
3. El resultado puede producir acarreamiento.

1 - Los tipos de las variables han de ser idénticos:

```
set clock = external;
void main (void)
{           int 4 w;
            int 4 y;
            unsigned 4 z;

            z = w + y;          // ERROR (z : = entero sin signo) ≠ ( entero con signo : = w, y)
                                // z : = signo y magnitud
                                // w, y : = complemento a dos
}
```

2 - El ancho de los operadores y el resultado ha de ser el mismo:

```
set clock = external;
void main (void)
{           unsigned int 4 w;
            unsigned int 3 x;
            unsigned int 4 y;

            w = 0b0110;
            x = 0b001;

            y = w + x ;                 // (1) ERROR, ( |w| = 4 ) ≠ ( 3 = |x| )

            y = w + ( 0 @ y );          // (2) concatenación de un "0" a la variable "y",
                                        // para aumentar el ancho de 3 a 4.
            y = w + adju(y, width(w));  // (3) produce el mismo resultado que (2)
}
```

(1)

Variable	Valor				Ancho
w	0	1	1	1	4
x		1	0	0	3

(2)

Variable	Valor				Ancho
w	0	1	1	0	4
0@x	0	0	0	1	1+3
y	0	1	1	1	4

3- El resultado puede producir acarreamiento:

```
set clock = external;
void main (void)
{           unsigned int 8 w,x, y;
            unsigned int 9 z;
            unsigned int 16 mul;

            x = 0xFF;
            y = 0x01;

            w = x + y ;                 //  (1) w = 0x00
            z = (0@x) + (0@y);          //  (2) z = 0x100
            y = 4;
            mul = (0@x) * (0@y);        // (3) mul = 0x00FF * 0x0004 = 0x03FC
}
```

(1)

	x	1	1	1	1	1	1	1	1
+	y	0	0	0	0	0	0	0	1
	w	0	0	0	0	0	0	0	0

(2)

	0@x	0	1	1	1	1	1	1	1	1
+	0@y	0	0	0	0	0	0	0	0	1
	z	1	0	0	0	0	0	0	0	0

(3)

	0@x	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
*	0@y	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	z	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0

5.6.2.4.3 Relacionales

Operador	Significado
==	Igual que
!=	Distinto que
<	Menor que
>	Mayor que
<=	Menor o igual que
>=	Mayor o igual que

Cada uno de los operadores anteriores, compara dos expresiones del mismo tamaño y devuelve un unsigned int 1, "1" si es cierto, "0" en cualquier otro caso.

```
unsigned 8 w, x, y, z;
w = x + (y >z);                 // ERROR  |w| = |x| = |y| = |z| = 8 ≠ 1 = |(y >z)|
w = x + (0 @ (y > z));
```

5.6.2.4.4 Lógicos

Operador	Significado
&&	Igual que
||	Distinto que
!	Menor que

Cada uno de los operadores anteriores, permite combinar operadores relacionales o lógicos, es decir, combina valores de cierto o falso. Los operandos son del tipo unsigned int 1, y resultado es un unsigned int 1, "1" si es cierto, "0" en cualquier otro caso.

```
if (x || y > w) w = 0;          // |x|=1
if (x != 0 || y > w) w = 0;     // |x|>=1
```

5.6.2.4.5 Bit a bit (bitwise)

Operador	Significado
&	And
|	Or
^	Or exclusiva
~	Not

Los operadores lógicos bitwise necesitan que el ancho y el tipo de los operandos y el resultado sean el mismo. Estos operadores devuelven como resultado una variable del mismo tipo y ancho que los operadores, ya que las operaciones se realizan bit a bit.

```
set clock = external;

void main (void)
{       unsigned int 6 w;
        unsigned int 6 x;
        unsigned int 6 y;
        unsigned int 6 z;

        w = 0b101010;
        x = 0b011100;

        y = w & x;          // y=0b001000
        z = w | x;          // z=0b111110
        w = w ^ ~x;         // w=0b001001
}
```

5.6.2.5 Estructuras de control

5.6.2.5.1 Estructura if-else

- Las condiciones se evalúan en 0 ciclos de reloj.
- Exactamente igual que en C.
- Se puede usar la instrucción delay en la parte else para equilibrar el tiempo de ejecución.

Si la expresión es cierta (a==0) se ejecuta el bloque de código (puede ser una sentencia (a++) o un conjunto de sentencias {…}) que hay debajo de la expresión.

Si es falsa, se ejecutará el bloque de código que hay después. En caso que aparezca la palabra reservada “else”, será el bloque de código que tiene debajo (delay), si no aparece la palabra “else”, el programa continuará saltándose el bloque de código que esta debajo de la expresión.

```
if (a == 0)
        a++;
else
        delay;
```

En el siguiente ejemplo, no hay bloque else. Por lo que el tiempo de ejecución es distinto, este depende del valor de la condición.

Ciclos	Caso 1	Ciclos	Caso 2
1	a=0;	1	a=3;
1	If (a == 0) a++;	0	if (a == 0) a++;
Total	2 ciclos	Total	1 ciclo

En el siguiente ejemplo, el tiempo de ejecución es idéntico independientemente del valor de la condición, ya sea cierta o falsa.

Ciclos	Caso 1	Ciclos	Caso 2
1	a=0;	1	a=3;

1	if (a == 0)
	a++;
	else
	delay;
		1	if (a == 0)
			a++;
			else
			delay;

Total	2 ciclos	Total	2 ciclos

5.6.2.5.1.1 Construcción de expresiones condicionales

Formato: **Expresión1 ? Expresión2 : Expresión3**

Permite realizar una asignación dependiendo del valor de la primera expresión. Si la primera expresión es cierta se toma el valor de la segunda expresión, si es falsa se toma la tercera expresión. Por ejemplo:

```
x = (y > z) ? y : z;

// Es equivalente a:
if (y > z) x = y;
else x = z;
```

5.6.2.5.2 Estructura switch

En la sentencia **switch**, se compara una a una, el valor de una variable o el resultado de evaluar una expresión, con un conjunto de constantes.

```
switch (Expresión)
{    case Constante:
            Sentencia
            break;
     .....
    default:
            Sentencia
            break;
}
```

Cuando coinciden se ejecuta el bloque de sentencias que están asociadas con dicha constante. El bloque de sentencias no está entre llaves, sino que empieza

con la palabra reservada **case** y termina con **break**. Si no se encuentra coincidencia, se ejecuta la sentencia **default**, si es que está presente en el código.

```
switch (x){
        case 10:            // Bloque 10
                a = b;
        case 11:            // Bloque 11
                c = d;
                break;      // Fin Bloque 10 y 11
        case 12:            // Bloque 12
                e = f;
                break;      // Fin Bloque 12
}
```

Si (x==10), a=b y c=d.Si (x==11), c=d. Si (x==12), e=f.

5.6.2.5.3 Estructura do-while

La sentencia o bloque de código que esta a continuación de "do", se repetirá 1 ó N veces (N≥1), mientras la condición evaluada sea cierta.

```
x=0;

do{     y=y+5;
        x=x+1;
}while(x!=45);
z=0;                        // x=45, y = y + 45*5, z=0
```

5.6.2.5.4 Estructura while

La sentencia o bloque de código que esta a continuación de while, se repetirá 0 ó N veces (N≥0), mientras la condición evaluada sea cierta.

```
x=0;

while(x!=45){
        y=y+5;
        x=x+1;
}
z=0;                        // x=45, y = y + 45*5, z=0
```

5.6.2.5.5 Estructura for

- Sintaxis

El cuerpo del bucle se ejecutará N veces, siendo N≥0.

```
for (Inicialización; Testeo; Incremento)
        Cuerpo
```

- Todas las expresiones son opcionales

  Puede no tener alguno de los 3 elementos (Inicialización; testeo; incremento):

```
for ( ; x>y ; x++ ){
        a = b;
        c = d;
}
```

  Puede tener varios, (entre {…;…;}) :

```
for ({ x=0; y=23;} ; x < 20; {x+=1; z*=2;} ){
        y = y + 2;
}
```

- **Inicialización,** tarda un ciclo de reloj.
- **Testeo,** se evalúa antes de cada iteración del **cuerpo.**
- La expresión **incremento** tarda un ciclo de reloj al final de cada ejecución del **cuerpo.**
- for no se recomienda para un uso general:
  - Usar while o do…while

    Equivalente a:

```
Inicialización;
while (Testeo){
        Cuerpo;
        Incremento;
}
```

  - El **Incremento** puede hacerse siempre en paralelo con el **cuerpo**

5.6.2.6 Asignación de estructuras de control a hardware

- Cada bloque tiene una señal de entrada de Inicio (Start) y una de salida de Fin (Finish).
- La señal Start proviene del bloque que acaba de concluir.
- La función principal (main) tiene su propia señal Start.
- La señal Start se pone a valor alto durante un ciclo de reloj cuando se ejecuta la instrucción.
- Cada nube de los diagramas representa otra instrucción que consume 0 o más ciclos de reloj para ejecutarse.

5.6.2.6.1 Asignación

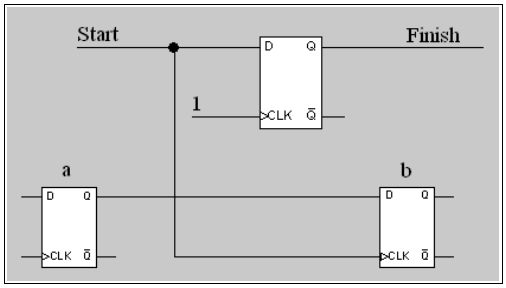


Ilustración 108: Síntesis a hardware, asignación

```
a=b;
```

5.6.2.6.2 Ejecución secuencial

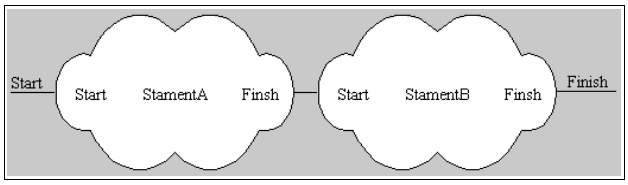


Ilustración 109: Síntesis a hardware, ejecución secuencial

```
seq
{
   StatementA;
   StatementB;
}
```

5.6.2.6.3 Ejecución paralela

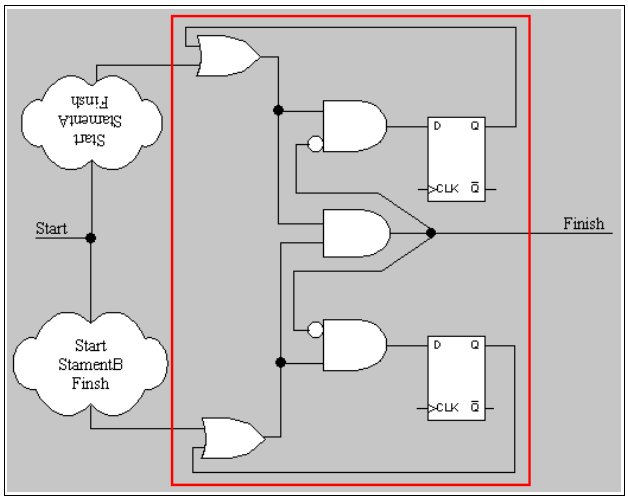
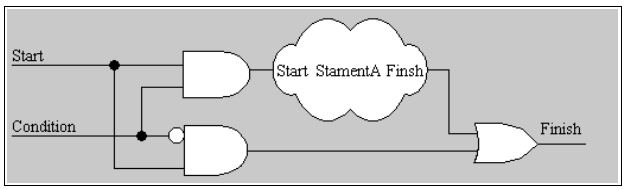


Ilustración 110: Síntesis a hardware, bloque sincronización del par

```
par
{
   StatementA;
   StatementB;
}
```

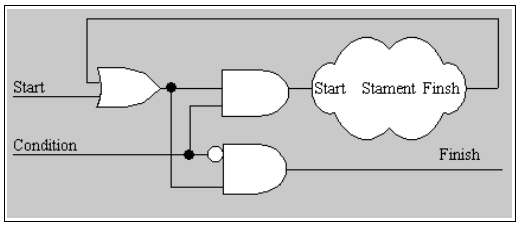
5.6.2.6.4 Instrucción if




```
if (condition)
        StamentA
```

Ilustración 111: Síntesis a hardware,instrucción if

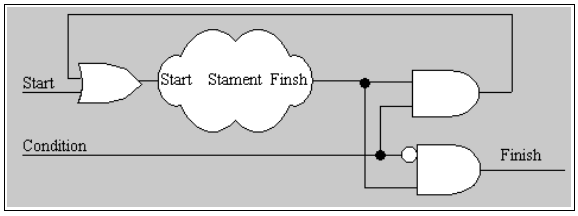
5.6.2.6.5 Bucles while




```
while(condition) Stament;
```

Ilustración 112: Síntesis a hardware, bucles while

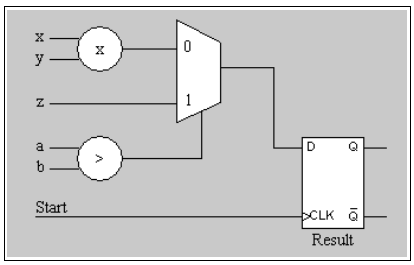
5.6.2.6.6 Bucles do…while




```
Do{
        Stament;
}while(condition);
```

Ilustración 113: Síntesis a hardware, bucles do ... while

5.6.2.6.7 Construcción de expresiones condicionales




```
Result = a > b ? x * y : z;
```

Ilustración 114: Síntesis a hardware,expresiones condicionales

5.6.2.7 Sincronización y Comunicación

- Muchos programas tienen procesos independientes ejecutándose en paralelo.
- Suelen necesitar comunicación y sincronización entre sí
- Suelen necesitar compartir recursos como funciones y RAMs
- Este tipo de código puede ser complicado y difícil de escribir
- Handel-C proporcionada dos características que permiten resolver estos problemas de forma más sencilla
  - Los canales para comunicación entre procesos
  - Los semáforos para controlar el acceso a secciones críticas

5.6.2.7.1 Canales

- Es una comunicación bloqueada entre dos secciones de código
  - Ambos extremos están bloqueados hasta que el otro esté listo

```
chan unsigned 8 ChannelA;       // ChannelA es un canal de 8 bits
unsigned 8 VarX;

ChannelA ! 3;                   // envía un 3 al canal ChannelA
ChannelA ? VarX;                // lee del canal ChannelA sobre la variable VarX
```

- La comunicación entre canales consume 1 ciclo de reloj
- Sólo se puede leer y escribir una vez en un canal en paralelo
  - La simulación puede detectar accesos en paralelo a canales
- El ancho de un canal se puede inferir igual que con variables normales
- Sólo se puede usar una anchura 0 con propósito de sincronización
- Los canales entre dominios de reloj se presentan más adelante
- chanin y chanout se usan para depuración
  - Por defecto Debug muestra valores en la Ventana de salida
  - Se pueden enlazar con archivos de texto

```
chan unsigned 8 ChannelA, ChannelB;
unsigned 8 VarX;
prialt{
        case ChannelA ! 3: DoSomething(); break;
        case ChannelB ? VarX: DoSomethingElse(); break;
        default: Otherwise(); break;
}
```

- Sintaxis similar a la instrucción switch
- Selecciona un evento de canal
  - El primero listo se lee o se le escribe
- El orden de prioridad es de arriba abajo
  - Cuanto más casos mayor profundidad de lógica
- Se bloquea salvo que haya un caso por defecto
- La comunicación del canal consume un ciclo de reloj
- No se permite la caída en cascada entre casos – se debe usar break en todos
- No se puede tener el mismo canal en dos casos

5.6.2.7.2 Semaforos

```
sema MySema; // MySema es un semáforo
while(trysema(MySema) == 0) // Espera hasta que pueda cogerlo
delay;
CriticalSection(); // Ejecuta la sección crítica
releasesema(MySema); // Libera el semáforo
```

- sema es un tipo
  - Los semáforos no tienen anchura o signo
- trysema() verifica y toma el semáforo si está disponible
  - Es una expresión, luego no consume ciclos de reloj
  - Devuelve 1 si tiene éxito
  - Implementa un bucle hasta que devuelva 1 al bloque
- releasesema() libera el semáforo
  - Consume un ciclo de reloj: está libre para ser cogido en el siguiente ciclo de reloj
  - No se puede llamar en paralelo con la última instrucción de la sección crítica
  - Se puede llamar en el mismo ciclo de reloj en que se toma
- Se debe usar un semáforo para proteger cada recurso
  - Bien se declara globalmente o se pasa como argumento a cada proceso que comparta el recurso
- No tienen prioridad definida – hay que programarlos de forma adecuada
  - No confiar en el orden
    - Se podría obtener un comportamiento diferente en simulación y en hardware
    - El comportamiento podría cambiar en función de cambios aparentemente no relacionados en el código
  - Tener cuidado con que los procesos no se queden sin datos
    - Un proceso podría soltar el semáforo y recuperarlo inmediatamente si tiene mayor prioridad que otros recursos en espera

5.6.2.8 Funciones y macros

Las funciones y macros encapsulan la implementación de una tarea en un bloque con nombre:

- Abstracción
  - Una función hace invisible y transparente su implementación, solamente se requiere su cabecera (prototipo o especificación).
  - La implementación de las cabeceras de las funciones proporcionadas por PAL son inaccesibles.
- Legilibilidad (Human readable)
  - La lectura de un código con funciones con un encabezamiento que describa perfectamente la tarea de una función, hace posible una lectura más rápida y un mejor entendimiento del código.
- Modularidad

  - Permite modificar el diseño, sin tener que cambiar drasticamente el código, tan solo es necesario reimplementar el cuerpo de las funciones, ya que se intentará mantener la cabecera de las funciones.
- Re-uso
  - Se pueden reutilizar porciones de código para nuevas aplicaciones.
- Compartición
  - El código encapsulado como función o macro puede ser fácilmente compartido entre procesos.

5.6.2.8.1 Funciones

```
int MyFunction(int a)
{
        static unsigned b = 1;
        b++;
        return a+b;
}
```

- Por defecto proporciona una funcionalidad compartida
  - Sólo se crea un bloque de hardware
  - Se multiplexan las entradas – puede tener un coste lógico extra

- Variables estáticas y automáticas
  - A las variables automáticas no se les puede dar un valor inicial
  - Sólo las variables estáticas mantienen sus valores entre llamadas a la función
  - Se dan valores iniciales sólo una vez al arrancar
  - Confiar en que las variables automáticas mantengan sus valores entre llamadas no es seguro

- Las funciones se puede declarar antes de su implementación
  - Ejemplo: int FunctionName([argumentos]);

- La llamada a una función en handel-C consume 0 ciclos.

5.6.2.8.1.1 Otras funciones

- Funciones Inline
  - inline ReturnType Name(ParameterList)
    - Se crea una nueva copia de la función para cada llamada
    - Muy similar a los macro procedimientos

- Arrays de funciones
  - ReturnType Name[n](ParameterList)
    - Se crean n copias de una función
    - Cada función del array es compartida
    - Se puede compartir cada copia entre varios procesos

5.6.2.8.1.2 Retorno de funciones

La asignación del valor de retorno consume un ciclo de reloj:

```
unsigned MyFunction(unsigned A){
        A++;                    // ciclo de reloj 1
        return A;               // ciclo de reloj 2
}

unsigned 8 a, b;
b = MyFunction(a);              // se asigna en el ciclo de reloj 2
```

Se puede pasar una referencia a la variable para no consumir el ciclo de reloj

La instrucción de retorno no puede estar en un bloque par

```
par{
        A++;
        return A;               // no permitido
}
```

5.6.2.8.1.3 Llamada múltiple a una misma función

Para que una misama función se pueda llamar múltiples veces en un mismo instante de tiempo, la función tiene que estar declarada como un array de funciones, por ejemplo si se desea implementar una función que devuelva el valor de una multiplicación, y que esta pueda ser llamada al mismo tiempo varias veces,

```
unsigned Mult(unsigned 8 A, unsigned 8 B){
        return A*B;
}

unsigned 8 a,b,c,d,e,f;
a = Mult(b,c);                  // llamada secuencial, no entran en conflicto
d = Mult(e,f);                  // ya que no se llaman al mismo instante de tiempo

par{
        a = Mult(b,c);          // ERROR
        d = Mult(e,f);          // ERROR
}
```

En el código anterior, se produce un error, ya que tan solo disponemos un bloque funcional para realizar el producto, y hay dos sentencias de código que intentan acceder al mismo tiempo al mismo bloque funcional. Para solucionar el problema anterior, podemos redefinirla funcion Mult:

```
unsigned Mult[2](unsigned A, unsigned B){
        return A*B;
}

par{
        a = Mult[0](b,c);
        d = Mult[1](e,f);
}
```

5.6.2.8.1.4 Parámetros de funciones

- Muy dependiente del tipo
  - Hay que asegurarse de que se pasa el tipo de variable esperado

- Llamada por valor
  - Lee la variable original en el primer ciclo de reloj
  - Trabaja con una copia en los siguientes ciclos de reloj
  - Las asignaciones a los argumentos no afectan a los valores originales
  - Puede añadir retrasos combinacionales en caminos que no se toman nunca
  - No se puede pasar por valor ni semáforos ni canales

- Llamada por referencia
  - No hay consumo extra en la copia (si no se modifica el puntero)
  - Los cambios en la variable pasada, afectan a la variable original
  - Nota: cambios en el valor del propio puntero NO modifican el puntero original – éste es pasado por valor

Ejemplo de Llamada por Valor:

```
unsigned Change(unsigned x){
        x++;                    // incrementa la copia del argumento a
        return x;               // devuelve el nuevo valor de la copia
}

static unsigned 8 a = 0, b;
Change(a);                      // pasa el valor de a, resultado final: a = 0, b = 1
```

Ejemplo de Llamada por Referencia:

```
unsigned Change(unsigned *x){
        (*x)++;                 // incrementa lo apuntado por x
        return *x;              // devuelve el valor de lo apuntado por x
}

static unsigned 8 a = 0, b;
b = Change(&a);                 // pasa un puntero a, resultado final: a = 1, b = 1
```

Ejemplo llamada por Referencia con punteros:

```
unsigned Change(unsigned *x){
        x++;                    // incrementa una copia del puntero
        (*x)++;                 // incrementa lo apuntado por x
        return *x;              // devuelve el valor de lo apuntado por x
}

static unsigned 8 a[2] = {1,2};
static unsigned *Ptr = a;       // Ptr apunta a a[0]
unsigned b;
b = Change(Ptr);
(*Ptr)++;                       // Incrementa lo apuntado por Ptr,
                                // Resultado final: a[0] = 2, a[1] = 3, b = 3
```

5.6.2.8.2 Macros

Existen dos tipos de macros:

- de **procedimientos**: grupos de instrucciones muy parecidas a las funciones
- de **expresiones**: son prototipos de expresiones complejas

Las macros de procedimientos y de expresiones:

- No tienen tipos en los argumentos
- Son Parametrizables
- Puedes ser recursivas en tiempo de compilación
- Crean un trozo de lógica hardware en cada llamada
- Funcionalidad adicional para el lenguaje del pre-procesador C
  - Pueden ser prototipadas como funciones
  - De sintaxis más clara

5.6.2.8.2.1 Macros de procedimientos

```
macro proc Change(A){
        A++;                    // incrementa A
}
```

- Produce una nueva copia de la lógica en cada llamada
- No hay tipos en los parámetros
  - Muy versátil
- No devuelven valor
  - Se pasa la variable que contendrá el valor como argumento
- Los macro procedimientos operan sobre las variables originales
- No hay consumo extra por la copia de datos
  - Es lo mismo que llamar por referencia
  - Sigue siendo una buena práctica pasar punteros para clarificar el significado
- Se pueden pasar expresiones como argumentos
- Se pueden pasar constantes como argumentos y usarlas para
  - Ancho
  - Dimensiones de Array/RAM

  Ejemplo:

```
macro proc Change(A)            // No hay tipo en argumentos, ni valor devuelto
{
        (*A)++;                 // Incrementa lo apuntado por A
}

unsigned 8 a;
Change(&a);                     // Pasa un puntero para clarificar el sentido
```

5.6.2.8.2.2 Macros de expresiones

Existen dos clases de macros de expresiones:

- No parametrizadas
  - macro expr Name = Expression;
  - Puede usarse como un #define
  - P. ej. macro expr Size = 8;
  - Se pueden usar para expresiones mucho más complicadas
  - P. ej. macro expr Sum = a+b;

- Parametrizadas
  - macro expr Name(Arguments) = expression;
  - Se puede emplear expresiones complicadas sobre los argumentos de entrada
  - No se comprueban los tipos de los argumentos
  - macro expr Add(a, b) = a+b;
  - unsigned 8 x, y, z;
  - z = Add(x,y);

Las macros de expresiones pueden evaluarse en tiempo de compilación o de ejecución:

```
macro expr Add(a,b) = a+b;
unsigned (Add(3,4)) x, y, z;

x = Add(y,z);
```

- Las expresiones evaluadas en tiempo de compilación se pueden usar para
  - Constantes
  - Anchos de palabra
  - Dimensiones de Array/RAM/ROM
  - Ejemplo de tiempo de ejecución: extensión de signo

```
macro expr SignExtend(X, Width) = ((signed (Width –width(X))((X)[width(X) - 1] ? ~0 : 0)) @
(signed) X;
```

5.6.2.9 Standard template library (stdlib.hch)

Platform Developer's Kit posee un librería estándar de Handel-C (stdlib.hcl) y un archivo de cabecera que contiene una colección de macros útiles. stdlib.hch no tiene nada que ver con la librería estándar de C stdlib.h. Para utilizar estas macros el archivo de cabecera tiene que estar incluido en programa escrito en Handel-C:

```
#include <stdlib.hch>
```

Además de incluir la línea anterior al código, también se tiene que añadir el módulo (stdlib.hcl) en la pestaña la propiedad linker del cuadro de configuración de proyecto (Project Settings) en DK GUI.

5.6.2.9.1 Constantes internas

La librería stdlib.hch contiene la definición de tres constantes, que permite disponer de un código más fácil de leer.

TRUE	1
FALSE	0
NULL	(void*)0

Por ejemplo:

```
int 8 x with { show=FALSE };
while (TRUE){
                    ...
}
if (a==TRUE){
          ...
}
```

5.6.2.9.2 Macros para la manipulación de bits

- **adjs**: Ajusta el ancho de una expresión con signo.
- **adju**: Ajusta el ancho de una expresión sin signo.
- **bitrev**: Expresión al revés.
- **copy**: Duplica una expresión.
- **lmo**: Buscar la posición del bit a 1 más significativo. (Utiliza lmo_nz() si la expresión no es cero).
- **lmz**: Buscar la posición del bit a 0 más significativo. (Utiliza lmz_no() si la expresón no todo son 1's).
- **population**: Numero de 1's en una expresión.
- **rmo**: Buscar la posición del bit a 1 menos significativo.
- **rmz**: Buscar la posición del bit a 0 menos significativo.
- **top**: Extrae los n bits más significativos.

5.6.2.9.2.1 adjs

Uso:	adjs(Expresión, Tamaño)	
Parámetros:	Expresión	Expresión a ajustar (entero con signo)
	Tamaño	Ancho a ajustar (una constante)
Devuelve:	Entero con signo de ancho igual a Tamaño.	
Descripción:	Incrementa o decrementa una expresión con signo a un ancho nuevo. Cuando se expande automáticamente se expande el signo (MSB). Cuando se reduce elimina los bits de mayor peso (MSB).	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 4 x;
int 5 y;
int 6 z;

y = 15;
x = adjs(y, Tamaño(x));          // x = -1
y = -4;
z = adjs(y, Tamaño(z));          // z = -4
```

5.6.2.9.2.2 adju

Uso:	adju(Expresión, Tamaño)	
Parámetros:	Expresión	Expresión a ajustar (enterero sin signo)
	Tamaño	Ancho a ajustar (una constante)
Devuelve:	Entero sin signo de ancho igual a Tamaño.	
Descripción:	Incrementa o decrementa una expresión sin signo a un ancho nuevo. Cuando se expande automáticamente se añaden ceros. Cuando se reduce elimina los bits de mayor peso (MSB).	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 4 x;
unsigned 5 y;
unsigned 6 z;
y = 14;
x = adju(y, Tamaño(x));          // x = 14
z = adju(y, Tamaño(z));          // z = 14
```

5.6.2.9.2.3 copy

Uso:	copy(Expresión, Contador)	
Parámetros:	Expresión	Expresión a copiar
	Count	Número de veces a copiar (constante)
Devuelve:	Expresión duplicated Count times.	
	Devuelve una expresión del mismo tipo que la expresión de entrada.	
	El ancho devuelto es igual a: C*W , donde C=Contador, W=Tamaño(Expresión).	
Descripción:	Duplica la Expresión varias veces.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 32 x;
unsigned 4 y;

y = 0xA;
x = copy(y, 8);          // x = 0xAAAAAAAA
```

5.6.2.9.2.4 lmo

Uso:	lmo(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	El primer bit que es ‘1’ desde la izquierda, o bién el ancho de la expresión si es cero. El ancho del valor devuelto es igual a log2ceil(Tamaño(Expresión)+1) bits.	
Descripción:	Encuentra la posición del primer bit a ‘1’ más significativo en una expresión.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
unsigned 4 y;

x = 27;
y = lmo(x);              // y = 4
x = 0;
y = lmo(x);              // y = 8
```

5.6.2.9.2.5 lmo_nz

Uso:	lmo_nz(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	El primer bit que es '1' desde la izquierda, o bién indefinido si la expresión si es cero. El ancho del valor devuelto es igual a log2ceil(Tamaño(Expresión)+1) bits.	
Descripción:	Encuentra la posición del primer bit a '1' más significativo en una expresión, ignorando el cero. lmo_nz() ocupa menos hardware que lmo().	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
unsigned 3 y;

x = 27;
y = lmo_nz(x);          // y = 4
x = 0;
y = lmo_nz(x);          // el valor de y es indefinido
```

5.6.2.9.2.6 lmz

Uso:	lmz(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	El primer bit que es '0' desde la izquierda, o bién Tamaño(Expresión) si todos son ceros. El ancho del valor devuelto es igual a log2ceil(Tamaño(Expresión)+1) bits.	
Descripción:	Encuentra la posición del primer bit a '0' más significativo en una expresión.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
unsigned 4 y;

x = 27;
y = lmz(x);             // y = 7
x = -1;
y = lmz(x);             // y = 8
```

5.6.2.9.2.7 lmz_no

Uso:	lmz_no(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	El primer bit que es ‘0’ desde la izquierda, o bién indefinido si todos son 1's. El ancho del valor devuelto es igual a log2ceil(Tamaño(Expresión)) bits.	
Descripción:	Encuentra la posición del primer bit a ‘0’ más significativo en una expresión, ignorando el caso que todos sean 1’s. lmz_no() ocupa menos hardware que lmz().	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
unsigned 3 y;

x = 27;
y = lmz_no(x);          // y = 7
x = -1;
y = lmz_no(x);          // el valor de y es indefinido
```

5.6.2.9.2.8 population

Uso:	population(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	El ancho de la salida es: log2ceil(Tamaño(Expresión +1)).	
Descripción:	Contar el número de bits a ‘1’ en una Expresión.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 4 x;
unsigned 3 y;

x = 0b1011;
y = population(x);      // y = 3
```

5.6.2.9.2.9 rmo

Uso:	rmo(Expresión)	
Parámetros:	Expresión	Expresión de entrada.
Devuelve:	El primer bit que es ‘1’ desde la derecha, o bién el ancho de la expresión si es cero. El ancho del valor devuelto es igual a (log2ceil(Tamaño(Expresión)+1) bits.	
Descripción:	Busca la posición del bit menos significativo a ‘1’ en una expresión.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
unsigned 4 y;

x = 26;
y = rmo(x);          // y = 1
x = 0;
y = rmo(x);          // y = 8
```

5.6.2.9.2.10 rmz

Uso:	rmz(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	El primer bit que es ‘0’ desde la derecha, o bién el ancho de la expresión si todos son 1’s. El ancho del valor devuelto es igual a (log2ceil(Tamaño(Expresión))+1) bits.	
Descripción:	Busca la posición del bit menos significativo a ‘0’ en una expresión.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 x;
unsigned 4 y;

x = 27;
y = rmz(x);          // y = 2
x = -1;
y = rmz(x);          // y = 8
```

5.6.2.9.2.11 top

Uso:	top(Expresión, Tamaño)	
Parámetros:	Expresión	Expresión de entrada
	Tamaño	Número de bits a extraer.
Devuelve:	El ancho del valor devuelto es igual al Tamaño.	
Descripción:	Devuelve los n bits más significativos de una expresión.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 32 x;
int 8 y;

x = 0x12345678;
y = top(x, Tamaño(y));          // y = 0x12
```

5.6.2.9.3 Macros aritméticas

- **abs**: Valor absoluto de una expresión.
- **addsat**: Suma saturada de dos expresiones, nunca mayor de $2^{Ancho+1}$-1.
- **decode**: Devuelve $2^{Expresión}$.
- **div**: Valor entero de Expresión1/Expresión2.
- **exp2**: Calcula $2^{Constante}$.
- **incwrap**: Incrementa un valor con sobrecarga con el segundo operando.
- **is_signed**: Determina el signo de una expresión.
- **log2ceil**: Calcula el $\log_2$ de un número redondeando al alza.
- **log2floor**: Calcula el $\log_2$ de un número redondeando hacia abajo.
- **mod**: Devuelve el resto de la Expresión1 dividida por la Expresión2.
- **sign**: Devuelve el signo de una expresión (0 positivo, 1 negativo).
- **subsat**: resta saturada de Expresión1-Expresión 2 (nunca <0).
- **signed_fast_ge, unsigned_fast_ge**: Devuelve uno si la Expresión1 es mayor o igual que la Expresión2, en otro caso devuelve cero.
- **signed_fast_gt , unsigned_fast_gt**: Devuelve uno si la Expresión1 es mayor que la Expresión2, en otro caso devuelve cero.
- **signed_fast_le , unsigned_fast_le**: devuelve uno si la Expresión1 es menor o igual que la Expresión2, en otro caso devuelve cero.
- **signed_fast_lt , unsigned_fast_lt**: Devuelve uno si la Expresión1 es menor que la Expresión2, en otro caso devuelve cero.

5.6.2.9.3.1 abs

Uso:	abs(Expresión)	
Parámetros:	Expresión	Expresión de entrada
Devuelve:	Un valor con signo del mismo ancho que expresión.	
Descripción:	Valor absoluto de una expresión.	
Requiere:	Archivo de cabecera: stdlib.hch	

	Librería de módulos: stdlib.hcl

Ejemplo:

```
int 8 x;
int 8 y;

x = 34;
y = -18;
x = abs(x);          // x = 34
y = abs(y);          // y = 18
```

5.6.2.9.3.2 addsat

Uso:	addsat(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 sin signo
	Expresión2	Operando 1 sin signo. Del mismo ancho que operando 1.
Devuelve:	Un valor sin signo del mismo ancho que Expresión1 y Expresión2.	
Descripción:	Devuelve la suma saturada de Expresión1 con Expresión2. El resultado no sera mayor que el valor máximo possible representable por el ancho de expresión 1 y expresión2.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 x,y,z;
x = 34;
y = 18;
z = addsat(x, y);    // z = 52
x = 34;
y = 240;
z = addsat(x, y);    // z = 255
```

5.6.2.9.3.3 decode

Uso:	decode(Expresión)	
Parámetros:	Expresión	Operando sin signo
Devuelve:	Valor sin signo de ancho $2^{\text{Tamaño (Expresión)}}$	
Descripción:	Devuelve $2^{\text{Expresión}}$	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 4 x;
unsigned 16 y;
x = 8;
y = decode(x);       // y = 0b100000000
```

5.6.2.9.3.4 div

Parámetros:	Expresión1	Operando 1
	Expresión2	Operando 2 sin signo. Del mismo ancho que Operando 1.
Devuelve:	Valor con mismo ancho que Expresión1 y Expresión2.	
Descripción:	Devuelve el valor entero de la división entera de Expresión1/Expresión2.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

5.6.2.9.3.5 exp2

Uso:	exp2(Constante)	
Parámetros:	Constante	Operando
Devuelve:	Constante de ancho Operando+1.	
Descripción:	Calcula $2^{Constante}$.Similar a decode, pero puede usarse con constantes de ancho indefinido.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 4 x;
unsigned (exp2(Tamaño(x))) y;   // y de ancho 16
```

5.6.2.9.3.6 incwrap

Uso:	incwrap(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1
	Expresión2	Operando 2. Del mismo ancho que Operando1
Devuelve:	Valor del mismo tipo y ancho que Expresión1 y Expresión2.	
Descripción:	Devuelve 0 si Expresión1 es igual a Expresión2, o en cualquier otro caso Expresión1+1.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 x;
x = 74;
x = incwrap(x, 76);          // x = 75
x = incwrap(x, 76);          // x = 76
x = incwrap(x, 76);          // x = 0
x = incwrap(x, 76);          // x = 1
```

5.6.2.9.3.7 is_signed

Uso:	is_signed(e)	
Parámetros:	e	Expresión no constante.
Devuelve:	1 si e posee signo, 0 si e no tiene signo.	
Descripción:	Determina el signo de una Expresión	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 a;
signed 6 b;
unsigned 1 Result;
Result = is_signed(a);          // Result == 0
Result = is_signed(b);          // Result == 1
```

5.6.2.9.3.8 log2ceil

Uso:	log2ceil(Constante)	
Parámetros:	Constante	Operando
Devuelve:	Una constante de valor ceiling($\log_2$(Constante)).	
Descripción:	Calcular $\log_2$ de un número con redondeo al alza. Muy útil para determinar el ancho de una variable requerida para almacenar un valor en particular.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned (log2ceil(5768)) x;    // x 13 bits de ancho
unsigned 8 y;

y = log2ceil(8);                // y = 3
y = log2ceil(7);                // y = 3
```

5.6.2.9.3.9 log2floor

Uso:	log2floor(Constante)	
Parámetros:	Constante	Operando
Devuelve:	Una constante de valor floor($\log_2$(Constante)).	
Descripción:	Calcular $\log_2$ de un número con redondeo a la baja.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 y;

y = log2floor(8);          // y = 3
y = log2floor(7);          // y = 2
```

5.6.2.9.3.10 mod

Parámetros:	Expresión1	Operando 1
	Expresión2	Operando 2. Del mismo ancho que Operando1.
Devuelve:	Valor del mismo ancho y tipo que Expresión1 y Expresión2.	
Descripción:	Devuelve el resto de la división Expresión1/Expresión2.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

5.6.2.9.3.11 sign

Uso:	sign(Expresión)	
Parámetros:	Expresión	Operando con signo.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Obtener el sino de una expresión. 0 si la expresión es positiva, 1 si la expresión es negativa.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 y;
unsigned 1 z;

y = 53;
z = sign(y);          // z = 0
y = -53;
z = sign(y);          // z = 1
```

5.6.2.9.3.12 subsat

Uso:	subsat(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 sin signo.
	Expresión2	Operando 2 sin signo. Del mismo ancho que Operando1.
Devuelve:	Valor sin signo del mismo ancho que Expresión1 y Expresión2.	

Descripción:	Devuelve el resultado de Expresión1-Expresión2. La resta está saturada y el resultado no sera inferior a 0.
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl

Ejemplo:

```
unsigned 8 x, y;
unsigned 8 z;

x = 34;
y = 18;
z = subsat(x, y);        // z = 16
x = 34;
y = 240;
z = subsat(x, y);        // z = 0
```

5.6.2.9.3.13 signed_fast_ge

Uso:	signed_fast_ge(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 con signo.
	Expresión2	Operando 2 con signo. Del mismo ancho que Operando1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es mayor o igual que la expresión2, 0 en cualquier otro caso.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
int 8 y;
unsigned 1 z;

x = 100;
y = -100;
z = signed_fast_ge(x, y);          // z = 1
x = -15;
y = -15;
z = signed_fast_ge(x, y);          // z = 1
```

5.6.2.9.3.14 unsigned_fast_ge

Uso:	unsigned_fast_ge(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 sin signo.
	Expresión2	Operando 2 sin signo. De mismo ancho que Operando 1.

Devuelve:	Entero sin signo de un bit de ancho.
Descripción:	Devuelve 1 si la expresión1 es mayor o igual que la expresión2, 0 en cualquier otro caso.
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl

Ejemplo:

```
unsigned 8 x, y;
unsigned 1 z;
x = 231;
y = 198;
z = unsigned_fast_ge(x, y);        // z = 1
x = 155;
y = 155;
z = unsigned_fast_ge(x, y);        // z = 1
```

5.6.2.9.3.15 signed_fast_gt

Uso:	signed_fast_gt(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 con signo.
	Expresión2	Operando 2 con signo. De mismo ancho que Operando 1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es mayor que la expresión2, 0 en cualquier otro caso.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x, y;
unsigned 1 z;
x = 100;
y = -100;
z = signed_fast_gt(x, y); // z = 1
x = -15;
y = -15;
z = signed_fast_gt(x, y); // z = 0
```

5.6.2.9.3.16 unsigned_fast_gt

Uso:	unsigned_fast_gt(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 sin signo.
	Expresión2	Operando 2 sin signo. De mismo ancho que Operando 1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es mayor que la expresión2, 0 en cualquier otro caso	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 x, y;
unsigned 1 z;
x = 231;
y = 198;
z = unsigned_fast_gt(x, y);        // z = 1
x = 155;
y = 155;
z = unsigned_fast_gt(x, y);        // z = 0
```

5.6.2.9.3.17 signed_fast_le

Uso:	signed_fast_le(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 con signo.
	Expresión2	Operando 2 con signo. De mismo ancho que Operando 1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es menor o igual que la expresión2, 0 en cualquier otro caso	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x, y;
unsigned 1 z;
x = -20;
y = 111;
z = signed_fast_le(x, y);          // z = 1
x = -15;
y = -15;
z = signed_fast_le(x, y);          // z = 1
```

5.6.2.9.3.18 unsigned_fast_le

Uso:	unsigned_fast_le(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 sin signo.
	Expresión2	Operando 2 sin signo. De mismo ancho que Operando 1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es menor o igual que la expresión2, 0 en cualquier otro caso	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 x, y;
unsigned 1 z;
x = 162;
y = 198;
z = unsigned_fast_le(x, y);        // z = 1
x = 155;
y = 155;
z = unsigned_fast_le(x, y);        // z = 1
```

5.6.2.9.3.19 signed_fast_lt

Uso:	signed_fast_lt(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 con signo.
	Expresión2	Operando 2 con signo. De mismo ancho que Operando 1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es menor que la expresión2, 0 en cualquier otro caso.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
int 8 x;
int 8 y;
unsigned 1 z;

x = -57;
y = -22;
z = signed_fast_lt(x, y);          // z = 1
x = -15;
y = -15;
z = signed_fast_lt(x, y);          // z = 0
```

5.6.2.9.3.20 unsigned_fast_lt

Uso:	unsigned_fast_lt(Expresión1, Expresión2)	
Parámetros:	Expresión1	Operando 1 sin signo.
	Expresión2	Operando 2 sin signo. De mismo ancho que Operando 1.
Devuelve:	Entero sin signo de un bit de ancho.	
Descripción:	Devuelve 1 si la expresión1 es menor que la expresión2, 0 en cualquier otro caso.	
Requiere:	Archivo de cabecera: stdlib.hch Librería de módulos: stdlib.hcl	

Ejemplo:

```
unsigned 8 x;
unsigned 8 y;
unsigned 1 z;

x = 162;
y = 198;
z = unsigned_fast_lt(x, y);        // z = 1
x = 155;
y = 155;
z = unsigned_fast_lt(x, y);        // z = 0
```

6 Diseño ESL de Celoxica

En este capítulo se describen los entornos de programación de Celoxica, concretamente DK design Suite, el utilizado en el proyecto. Además se describen los paquete, los requisitos, los componentes, el modelo de programación, y la arquitectura.

El diseño ESL que proporciona Celoxica esta centralido en la programación mediante el lenguaje C. Permite dividir las fases de diseño en varios equipos concentrados en una fase concreta de la implementación.

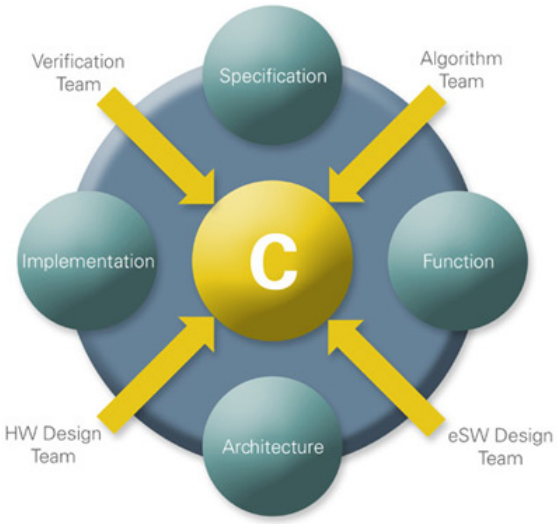


Ilustración 115: Diseño ESL de Celoxica

A partir de la especificación de la aplicación (Specification) se llega a la solución implementada en hardware (Physical Design), pasando por los tres niveles de ESL.

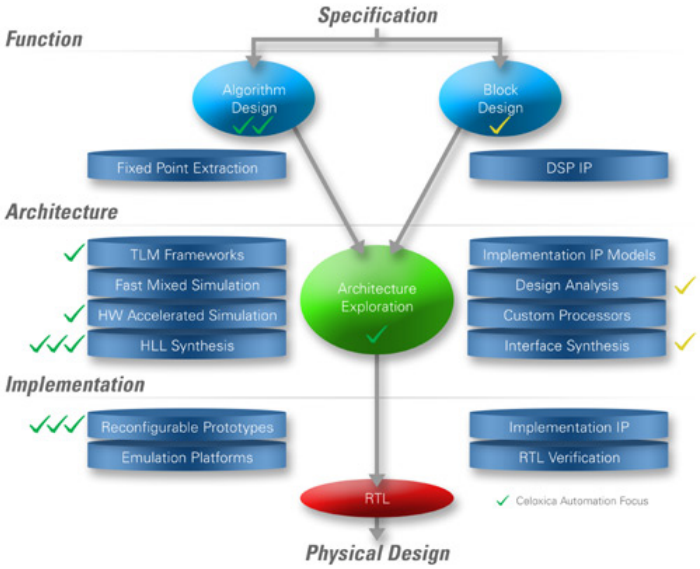


Ilustración 116: Tres niveles de ESL en Celoxica

Los tres niveles en ESL:

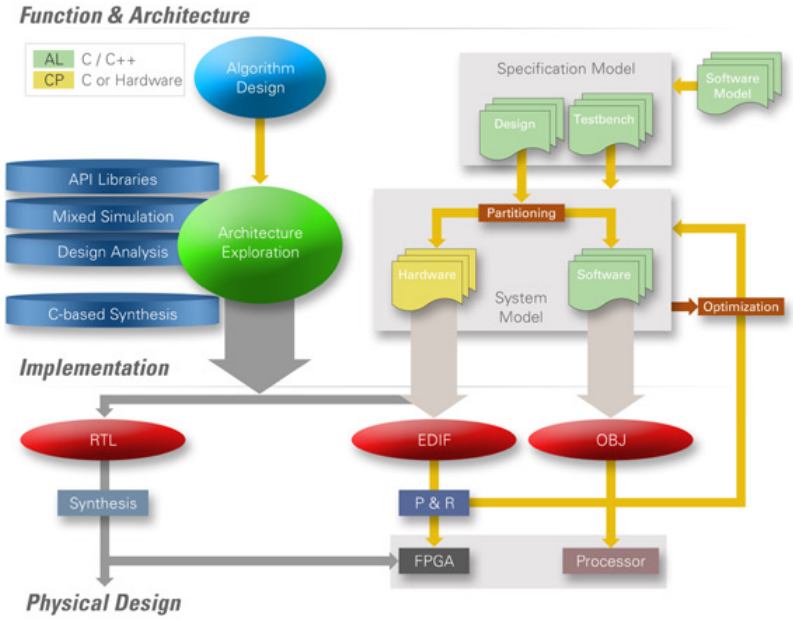


Ilustración 117: Tres niveles ESL de celoxica con detalle

Los lenguajes mediante los cuales se realiza la programación en C/C++ son Handel-C y System-C:

<table>
<tr><td colspan="2">Handel-C</td><td colspan="2">System-C</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td colspan="2" rowspan="7"></td><td colspan="2">Canales estándar para varios MOC</td></tr>
<tr><td colspan="2">Kahn Process Networks, Static Dataflow</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">Primitivas para los canales</td></tr>
<tr><td colspan="2">Signal, Timer, Mutex, Semaphore, Fifo, etc …</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2"></td></tr>
<tr><td colspan="2">Librerías principales</td><td colspan="2">Librerías principales</td></tr>
<tr><td colspan="2">TLM (PAL/DSM), Punto Fijo y Flotante</td><td colspan="2">SCV, TLM, Maestro/Esclavo</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td>Núcleo del Lenguaje</td><td>Tipos de datos</td><td>Núcleo del Lenguaje</td><td>Tipos de datos</td></tr>
<tr><td>Par{…} seq{…}, Interfaces, Canales, Manipulación a nivel de Bit, RAM y ROM, Asignación en un solo ciclo,</td><td>Vectores de Bit y Bits, Tamaño flexible para los enteros, Señales</td><td>Módulos, Puertos, Procesos, Eventos Interfaces, Canales, Eventos conducidos por señales del núcleo</td><td>Vectores de 4 valores lógicos, Vectores de Bit y Bits, Tamaño flexible para los enteros, Punto Fijo, Tipos C++ definidos por el usuario</td></tr>
<tr><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td colspan="2">Lenguaje Estándar C ANSI/ISO</td><td colspan="2">Lenguaje C/C++</td></tr>
</table>

6.1 Basado en SystemC

6.1.1 Paquetes de desarrollo

6.1.1.1 Agility Compiler

Agility es una herramienta para sintetizar de SystemC a FPGA, o de SystemC a RTL-VHDL/Verilog. Agility puede sintetizar un sistema completo de hardware, conteniendo cualquier número de módulos jerárquicos con cualquier número procesos sc_thread y sc_cthread (hilo cronometrado - clocked thread).

Soporta el tipo de nivel alto sc_fifo (no incluido en el Subconjunto Sintetizable), para facilitar el diseño, la depuración y la síntesis en diseños con múltiples dominios de reloj.

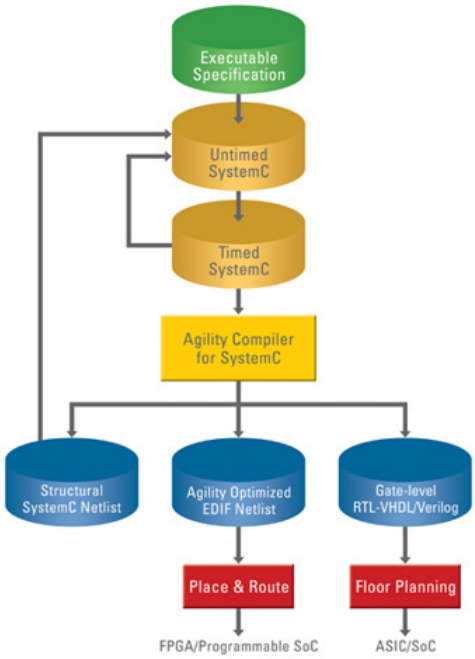


Ilustración 118: Agility Compiler

También poporciona soporte en la fase de compilación para tipos dobles y floats, dando soporte a la biblioteca math.h de C++. Se pueden crear inicializaciones complejas y estados de reset sin tener que manipular el código fuente a través de otra herramienta.

6.1.2 Pre-requisitos

Hardware:

- 500MBytes de disco duro
- 512MBytes RAM
- Unidad CD-ROM
- resolución de tarjeta gráfica a 1024x768 (recomendado)
- Tarjeta Ethernet (para la licencia y recomendado para PALSim)
- Tarjeta Sonido Entrada/Salida (recomendado para PALSim)

Software:

- MsWindows XP SP2
- Microsoft VC++ 7.1 (igual o superior), o Cygwin win32 (g++)
- Para FPGAs de Xilinx, se puede excojer:
  - Xilinx ISE 7.1i + SP4
    - Fundation
      - Paca RC203E (XC2V3000-4-FG676)
    - Base X
    - WebPack
  - Xilinx ISE 8.1i
    - Fundation
      - Placa RC203E (XC2V3000-4-FG676)
    - Webpack
- Celoxica Agility Compiler
- Para Xilinx MicroBlaze y Virtex II Pro PSL & soporte DSM:
  - Xilinx EDK 7.1i
- Para la co-simulación con ModelSim
  - Model Technology ModelSim SE o posterior
- Para utilizar MATLAB/SIMULINK S-Function:
  - MATLAB/SIMULINK

6.2 Basado en Handel-C

6.2.1 Paquetes de desarrollo

Bloque	DK Design Suite	Nexus-PDK	Platform Development Package
Cosimulación RTL con ModelSim	Opcional	Opcional	No
Cosimulación con Matlab	Si	Si	No
Cosimulación con SystemC	Si	Si	No
Soporte Procesador Softcore (DSM) para MicroBlaze de Xilinx	Si	Si	Opcional
Soporte Procesador Hardcore (DSM) para Xilinx VII Pro	Opcional	Opcional	Opcional
Soporte para deseño con PAL Sim	Si	Si	Si
Soporte para deseño DSM Sim	Si	Si	Opcional
RC200	Si	No	Opcional
RC2000	Si	No	Opcional
Soporte para placas de desarrollo con Altera NIOS	Si	No	Opcional
RC250	Si	No	Opcional
RC2000Pro	Si	No	Opcional
Soporte para tarjetas Altera NIOS II en los kits Stratix Edition y Stratix professional Edition	Si	No	Opcional
RC100	Si	No	No
PixelStreams	Opcional	No	No

6.2.1.1 Nexus PDK

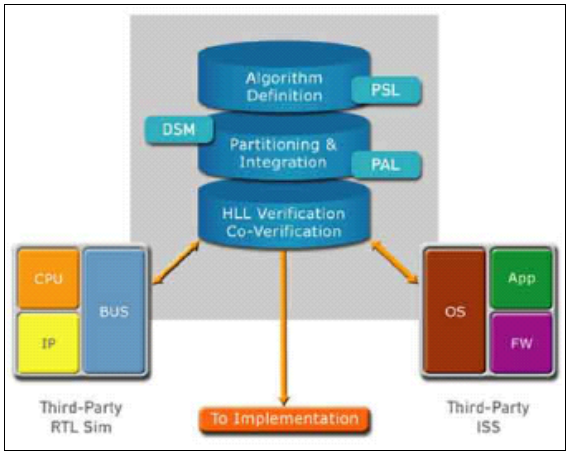


Ilustración 119: Nexus PDK

Nexos-PDK es una herramienta de Celoxica para realizar diseños de sistemas empotrados de una forma rápida y sencilla. Con esta herramienta se puede realizar conjuntamente el codiseño de hardware y software.

El codiseño de software esta compuesto por un sistema operativo (OS) que realiza el control del sistema, y permite acceder a la arquitectura y a los periféricos, de una forma transparente al programador de la aplicación. El firmware (FW) es el interfaz por el que se establecen, modifican o consultan los parámetros del sistema. Y por ultimo la aplicación (App), realiza un proceso o tarea determinado sobre el sistema diseñado.

El codiseño de Hardware esta compuesto por una CPU o unidad de control sobre la que se ejecutan las instrucciones del sistema. El flujo de datos pasa por las líneas del BUS comunicando todos los periféricos (IP) con la unidad de procesado (CPU).

Los tres niveles de diseño ESL utilizando Nexos-PDK son:

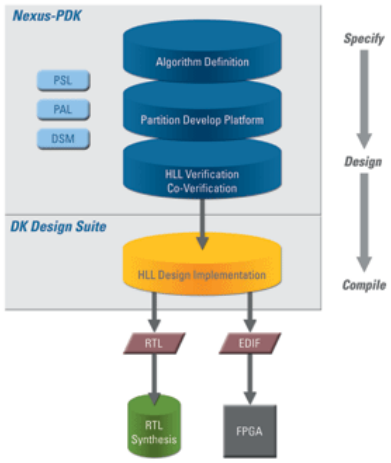
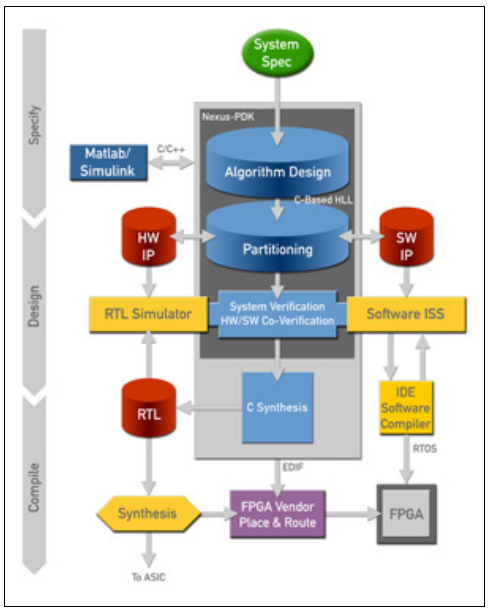


Ilustración 120: Tres niveles en diseño ESL mediante Nexus-PDK

6.2.1.2 DK Design Suite



DK (Design Kit, kit de desarrollo) es un Entorno Integrado de Desarrollo (IDE) para diseños basados en C que proporciona herramientas de diseño, simulación y exploración a través a la síntesis.

Ilustración 121: Tres niveles en diseño ESL mediante DK

6.2.1.3 Platform Development Package

Versión reducida de las herramientas de desarrollo de Celoxica. Se reciben al adquirir una placa de desarrollo de Celoxica. Se trata de una versión reducida del entorno de desarrollo DK design Suite, que solamente incorpora las APIs específicas para desarrollar con la placa adquirida.

6.2.2 System Level APIs de Celoxica - Platform Developer's Kit

PDK es un conjunto de librerías y herramientas que se integran en los entornos de programación de Celexica, tanto DK Design Suite como Nexos-PDK. PDK proporciona tres capas de APIs a nivel de sistema en diseños ESL, esta clasificación esta establecida según su funcionalidad:

- ***Platform Support Libraries (PSL)***
  - PSL proporciona acceso a los periféricos mediante drivers.
  - Contiene implementados los drivers para plataformas de Celoxica.
  - Proporcina las cabeceras de las funciones para modificar los drivers por si se cambian los periféricos.
  - Permite implementar nuevos drivers.

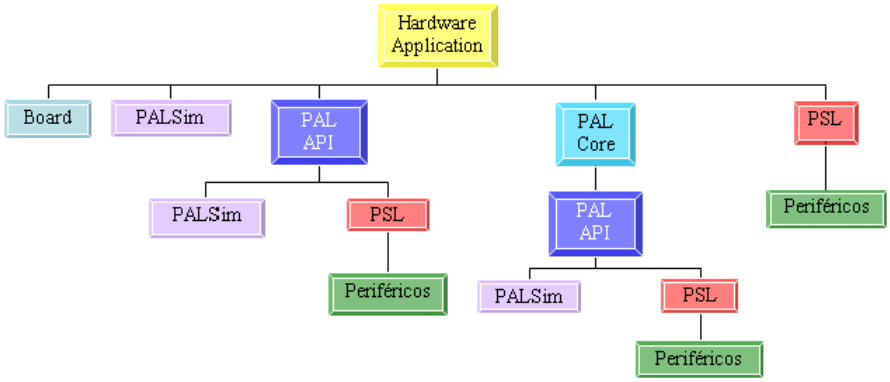
- ***Platform Abstraction Layer (PAL)***
  - Permiten realizar la programación de las aplicaciones independientemente de la implementación de los drivers (PSL).
  - Proporciona interfaces fijas.
  - Son aconsejables para poseer un código en Handel-C portable.

- ***Data Stream Manager (DSM)***
  - DSM proporciona integración entre procesadores/DSPs y FPGAs/PLDs, ya sea en un dispositivo integrado (hard-cores) o implementado (soft-cores).

Cada una de estas tres áreas de funcionabilidad proporciona:

- **Simulación** - Simulación independiente del hardware.
- **Kit** - Componentes dominantes y/o plantillas para permitir el desarrollo de las nuevas prestaciones.
- **Plataforma**– Implementaciones de DSM, PAL y PSL para satisfacer los componentes de una plataforma o entorno específico de desarrollo.
- **IP cores** – Implementaciones de nuevas funcionalidades, demostraciones o ejemplos.

	Simulación	Kit	Implementación de plataformas	IP Cores
DSM	DSM Sim	DSM plantillas y ejemplos	Plataformas específicas DSM	Basados en IP DSM y ejemplos
PAL	PAL Sim	PAL plantillas y ejemplos	Plataformas específicas PAL	Basados en IP PAL
PSL	Co-simulación: ISS, HDL, Modelado	Drivers a nivel de dispositivo, plantillas y ejemplos	Plataformas específicas PSL	IP específicos: placa y dispositivos



6.2.2.1 Platform Support Libraries (PSL)

PSL es un conjunto de librerías que proporcionan las implementaciones y cabeceras de funciones, para varios módulos de Entrada/Salida para plataformas de hardware específicas, permitiendo al desarrollador implementar o utilizar drivers para controlar los periféricos. De este modo, si se cambia de plataforma, solo se tiene que cambiar el driver (implementación).

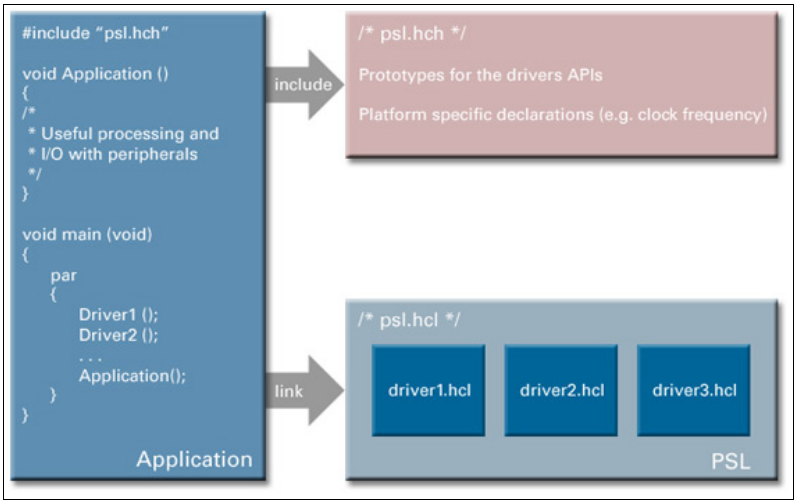


Ilustración 122: Platform Support Libraries (PSL)

PSL incorpora implementados los drivers para las plataformas RC de Celoxica. Además da soporte a los soft-cores MicroBlaze y Nios, a los hard-cores de PowerPC, y a Rocket I/O.

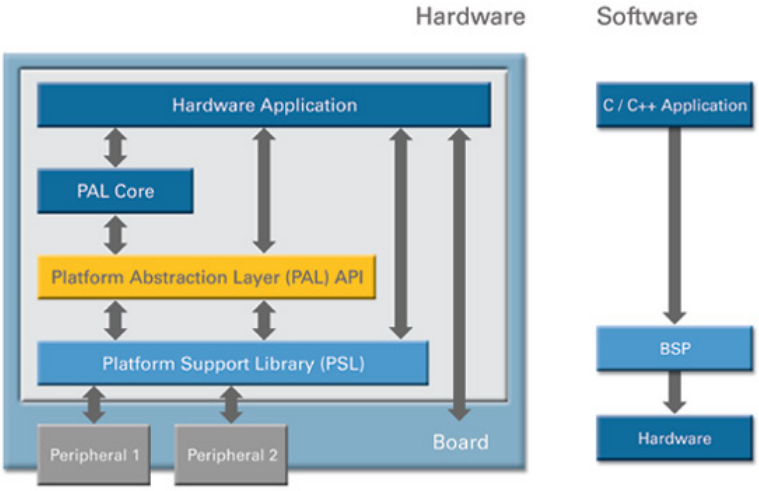
Módulos de E/S en la plataforma de trabajo RC-203:

- Synchronous Static RAM
- Salida LED
- Display de 7 segmentos
- Puerto Paralelo
- RS232
- Ethernet
- Puerto PS/2
- Memoria Flash (Smart Media)
- Composite Video In
- Entrada S-Vídeo
- Salida S-Vídeo
- Salida VGA
- Entrada Audio
- Salida Audio
- Interruptores y botones
- Módulo BlueTooth
- Pantalla TFT
- Pantalla táctil

Además PSL da soporte a la integración de los siguientes cores procesadores:

- Procesador MicroBlaze
  - OPB bus
- Xilinx VII Pro
  - OPB Bus
  - PLB Bus

6.2.2.2 Platform Abstraction Layer (PAL) 1.3



PAL son un conjunto de APIs (Application Program Interfaces) desarrolladas por Celoxica y proporcionan interfaces entre módulos de estrada/salida (periféricos) y aplicaciones escritas en Handel-C.

Ilustración 123: Estructura general PAL

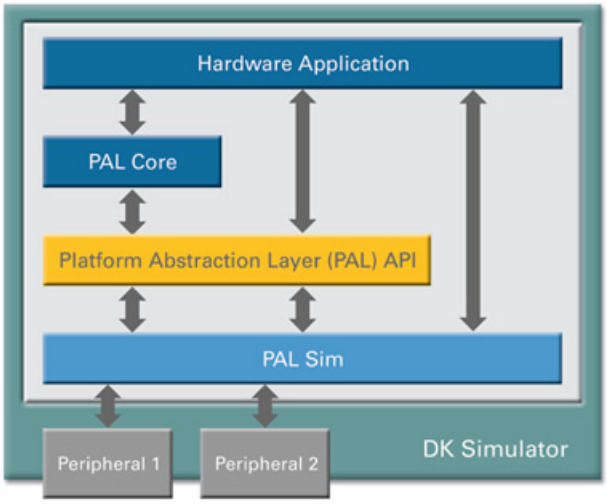
PAL proporciona el interfaz para los siguientes dispositivos:

- LEDs
- Displays de 7 segmentos
- Interruptores y botones
- Puertos de datos
- Puertos Paralelos
- Puertos serie RS-232
- Puertos PS2
- RAMs rápidas
- RAMs segmentadas (PL1 y PL2)
- RAMs lentas
- SDRAMs
- Dispostivos de almacenamiento en bloque (ej. Memoria flash)
- Salida de vídeo
- Entrada de vídeo
- Ethernet
- Entrada de audio
- Salida de audio

Las API's permiten cambiar de tarjeta/plataforma de desarrollo sin modificar de forma significativa el modo en el que se llaman a las funciones, ya que la implementación para cada tarjeta/plataforma de desarrollo la incorpora la PSL. PAL funciona bajo las siguientes tarjetas de desarrollo:

- Celoxic RC10, RC100, RC1000, RC200 y RC200E, RC203 y RC203E, RC250 y RC250E, RC300 y RC300E
- Celoxica RC2000 (ADM-XRC-II)
- Altera NiosII development board stratix edition (NDBS)
- Memec MV2P
- Celoxica PALSim Virtual Platform

6.2.2.2.1 Celoxica PALSim Virtual Platform



PALSim simula en un ordenador personal una plataforma hardware con una funcionalidad idéntica a las tarjetas de desarrollo. Mediante el uso de ventanas se puede observar la ejecución de una aplicación utilizando las API's de PAL.

Ilustración 124: Estructura DK Simulator (PALSim VP)

PALSim proporciona los siguientes tipos de interfaces:

- Memoria (including load from- and save to- disk)
- Puertos de datos (Desde y hacia un archivo de disco)
- Displays LEDs y de 7 segmentos
- Captura de vídeo (desde un fichero)
- Salida de vídeo
- Almacenamiento en Bloque
- Botones y interruptores
- Ethernet
- Audio
- Ratón
- Teclado

Al ser simulado se pueden verificar diseños sin necesidad de implementarlos bajo hardware, es decir sin la necesidad de utilizar una tarjeta de desarrollo con una FPGA. Al mismo tiempo partiendo de una simulación por ordenador, es posible diseñar aplicaciones independientes al hardware utilizado a posteriori.

6.2.2.2.2 PAL – cores 1.2

PAL incorpora además un conjunto de cores que permiten gobernar diferentes dispositivos. Los PAL cores que actualmente están implementados son:

- **Console**: Una librería genérica para controlar la consola de vídeo.
- **Framebuffer**: Una librería genérica para controlar frame buffers de 8 y 16 bits.

- **Mouse driver**: Una librería genérica para controlar ratones.
- **Keyboard driver**: Una librería genérica para controlar teclados.

6.2.2.3 Data Stream Manager (DSM)

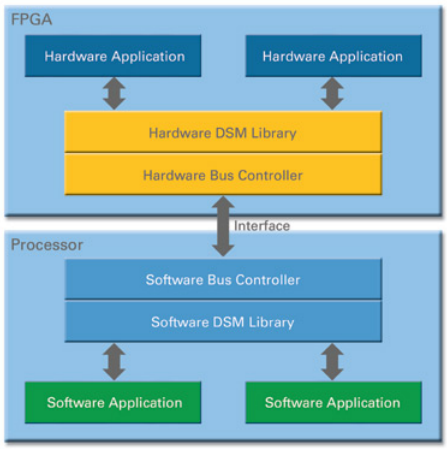


Ilustración 125: Data Stream Manager (DSM)

Data Streaming Manager (DSM) permite el co-diseño, tanto a nivel hardware: diseño/inclusión del procesador, diseño/inclusión de módulos adicionales, buses de sistema para interconectar los módulos con el procesador. Como también el diseño software: Sistema Operativo y aplicaciones escritas en C/C++. DSM proporciona la escalabilidad de este tipo de diseños especificando las librerías.

El API DSM, esta dividido en dos partes: Un API para programas software escritos en C, y una segunda API para harware escrito en C. Al mismo tiempo, los programas escritos con APIs de C, pueden utilizar porciones de código escritos en otros lenguajes, por ejemplo el software escrito en C, puede incluir código en ensamblador, en Fortran,…, y el hardware escrito en C puede incluir código escrito en VHDL.

6.2.3 Pre-requisitos

Hardware:

- 500MBytes de disco duro
- 512MBytes RAM
- Unidad CD-ROM
- Resolución tarjeta gráfica 1024x768 (recomendado)
- Tarjeta Ethernet (para la licencia y recomendado para PALSim)
- Tarjeta Sonido Entrada/Salida (recomendado para PALSim)

Software:

- MsWindows XP SP2
- Compilador Back End:
  - Simulación
    - Cygwin g++ 3.3.3, (gratuito) o bién
    - Microsoft Visual-C++ 2005 express (gratuito)
  - FPGAs de Xilinx, soportados:
    - Xilinx ISE 7.1i + SP4
      - Fundation
        - Paca RC203E (XC2V3000-4-FG676)
      - Base X
      - WebPack (gratuito)
    - Xilinx ISE 8.1i

        - Fundation
            - Placa RC203E (XC2V3000-4-FG676)
        - Webpack (gratuito)
    - Xilinx MicroBlaze y Virtex II Pro PSL & soporte DSM:
        - Xilinx EDK 7.1i
- Compilador Front End:
    - Celoxica DK 4.0 SP1 o posterior
    - Celoxica PDK 4.1 o posterior
        - Para usar las funcionalidades:
            - PSL
            - PAL
            - DSM
- Para la co-simulación con ModelSim
    - Model Technology ModelSim SE o posterior
- Para usar MATLAB/SIMULINK S-Function:
    - MATLAB/SIMULINK
- Para la co-simulación con SystemC:
    - SystemC 2.0.1 (incluido con Agility Compiler)

7 Entorno de desarrollo: DK Design suite

En este capítulo se describe el uso de la herramienta de DK Design Suite para realizar aplicaciones mediante FPGAs con el lenguaje de programación handel-C. Detallando las pasos para cada una de las fases del diseño, incluyendo la información necesaria para la construcción y configuración de un proyecto, así como los sucesivos pasos hasta llegar a la implementación de un diseño a una FPGA.

7.1 Etapas en la realización de un proyecto

Las etapas comunes en la realización de un proyecto simple son:

1. Creación de un nuevo proyecto
2. Realizar la configuración del proyecto
3. Añadir código fuente al proyecto
4. Añadir librerías
5. Configurar el proyecto para el debugger
6. Compilar el proyecto para el debugger
7. Utilizar el debugger y el simulador
8. Optimizar el proyecto/código
9. Compilar el proyecto para el dispositivo específico (target chip)
10. Exportar el archivo EDIF a la herramienta de place & route
11. Realizar place & route

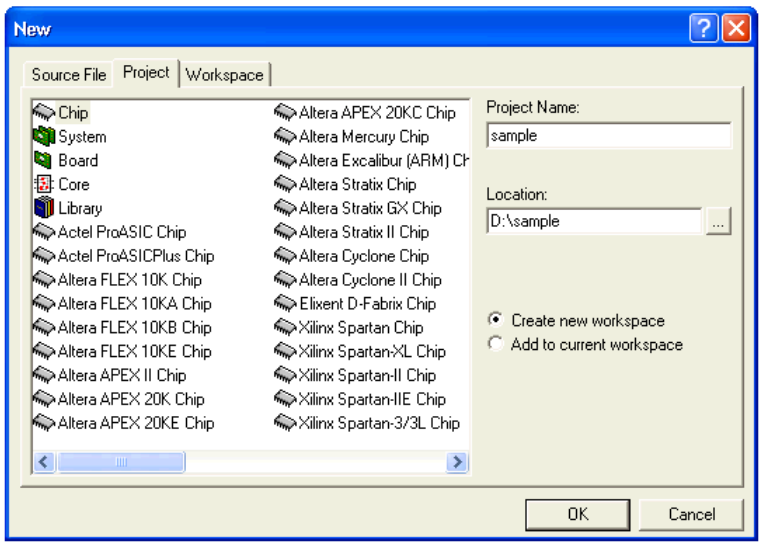
7.2 Creación de un nuevo proyecto

7.2.1 Creación de proyectos y relación con workspace

Un proyecto esta compuesto principalmente por un conjunto de directorios y ficheros. Un Workspace, entorno de trabajo, proporciona un envoltorio para agrupar las propiedades de uno o varios proyectos. El objetivo principal de Workspace es el acceso rápido y estructurado a los ficheros que componen un proyecto, añadiendo iconos a los diferentes tipos de archivos, acceso estructurado a los ficheros, y asistentes para modificarlos.

Antes de crear un proyecto nuevo, se puede crear/abrir un Workspace donde situar los proyectos nuevos o ya creados. Esta opción no es obligatoria pero es recomendable, ya que de este modo se tendrán los proyectos agrupados.

Para crear un proyecto nuevo, se seleccionará en la barra de menús la opción “File”, y a continuación “New”. Especificaremos el tipo de proyecto, el nombre (Project Name) y su ubicación (Location).



7.2.2 Tipos de poyectos

En un proyecto se disponen de forma agrupada un conjunto de archivos según su funcionalidad u objetivo. En DK-Design Suite, un proyecto es la compactación en forma de árbol de diferentes tipos de archivo de forma estructurada y ordenada, en varios subdirectorios y apartados.

Hay varios tipos de proyectos según si se quiere implementar toda la aplicación en un mismo proyecto, o si queremos implementar nuestra aplicación dividiéndola en subproyectos. Al mismo tiempo podemos incluir librerías o módulos ya implementados o crear nuevos. También se pueden implementar módulos genéricos que podrían servir para futuras aplicaciones.

Actualmente DK posee 6 tipos de proyectos:

- Chip
  - No está dirigido hacia un producto en particular.
  - No usará los recursos de ningún dispositivo específico.
  - No puede construirse como módulo genérico.
- Board
  - Permite hacer varios proyectos Chip un mismo proyecto.
  - Dirigido a chips específicos definidos dentro de un board.
  - No puede construirse como módulo genérico.
- System
  - Permite hacer varios proyectos Board un mismo proyecto.
  - Dirigido a chips específicos definidos dentro de un board.
  - No puede construirse como módulo genérico.
- Core
  - Trozo pequeño de código compilado para una arquitectura específica (FPGA), el cuál puede ser usado como parte de un diseño más grande.
  - No puede construirse como módulo genérico.
- Library
  - Código precompilado en Handel-C que puede ser reutilizado o distribuido.
  - Si está construido en modo Genérico puede ser reconstruido para ser dirigirlo a EDIF, VHDL o Verilog.
  - Estando construido en otro modo diferente al genérico sólo puede ser conectado con proyectos en ese formato.
- Pre-defined chip, system or board
  - Orientado a un dispositivo concreto.
  - Se realizarán optimizaciones para ese dispositivo.
  - Solo se podrá ubicar (Place&Route) en ese dispositivo.
  - No puede construirse como módulo genérico.

7.3 Realizar la configuración del proyecto

7.3.1 Configuración por defecto

A la agrupación de todos los parámetros que describen propiedades de un proyecto (project settings) se le denomina configuración (configuration). DK proporciona 6 configuraciones por defecto:

- Debug
- Release
- VHDL
- Verilog
- EDIF
- Generic

7.3.2 Crear nuevas configuraciones

El proceso de creación de una nueva configuración, consiste en copiar una configuración ya existente a otra nueva, copiando en ello el valor de los parámetros establecidos en la original a la nueva configuración. Posteriormente se realizarán los cambios oportunos a la nueva configuración. Los pasos son los siguientes:

1 Seleccionando Build>Configurations...
2 Cliquear en el botón añadir (add) en la pantalla de dialogo que aparece.
3 Introducir el nombre para la nueva configuración, y seleccione la configuración base para la nueva configuración i presione en aceptar (Ok).
4 Cerrar el cuadro.
5 Abrir las propiedades del proyecto (Project settings), seleccionar la nueva configuración y editar los campos necesarios.

Las configuraciones creadas por el usuario sólo están disponibles dentro del proyecto en el que fueron creadas. El número máximo de configuraciones por cada proyecto simple (en agrupaciones de proyectos, es cada uno de los proyectos por individual) es de 1024.

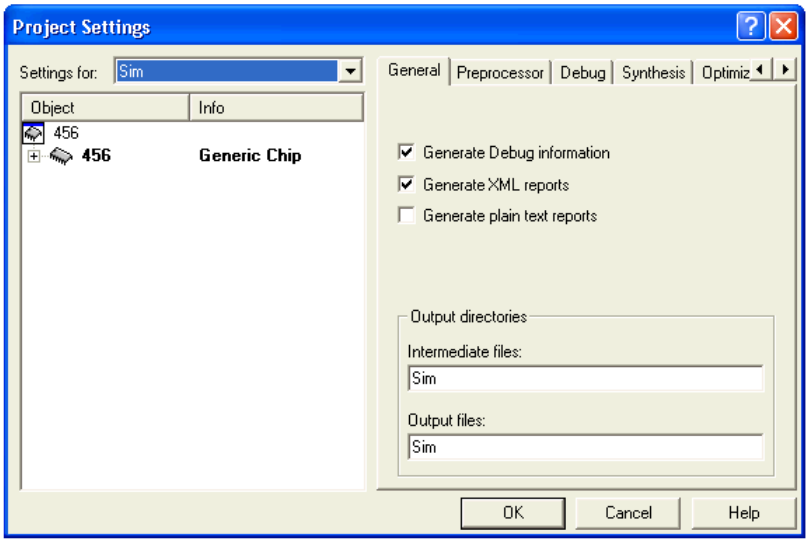
7.3.3 Modificar la configuración del proyecto

7.3.3.1 Propiedades del proyecto (Project settings)

Las pestañas de propiedades de proyecto son:

- General
- Preprocessor
- Debug
- Synthesis
- Optimization
- Chip
- Linker
- Build commands
- Library

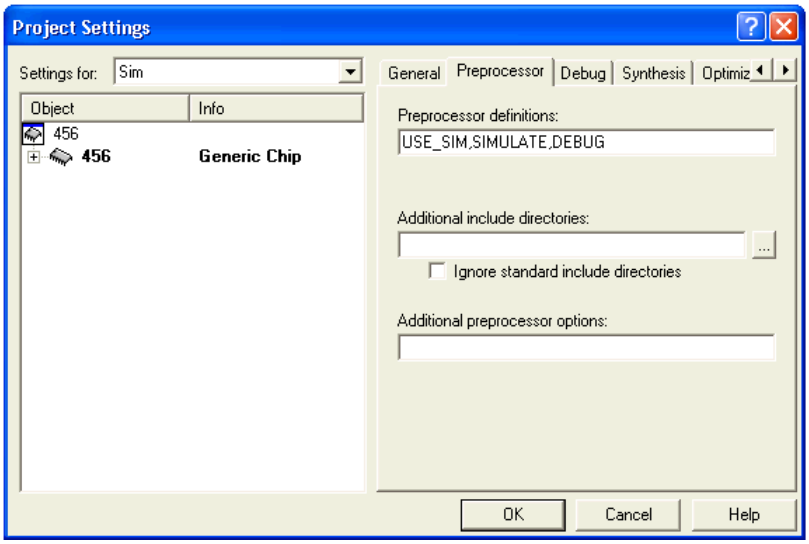
7.3.3.1.1 Pestaña: General



Diversos ajustes están disponibles para los proyectos y para archivos individuales.

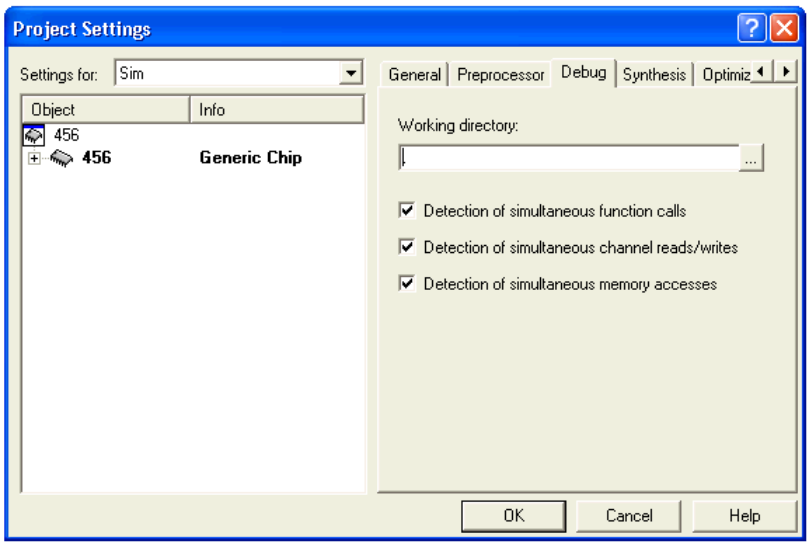
Elemento	Significado	Valor	Defecto
Generate debug information	Compilación con debugaje activado en el proceso de simulación. Solamente disponible en Debug, Release y Generic modes.	Seleccionar para activar	Seleccionado: para Debug. No seleccionado: para Release o Generic.
Always Use Custom Build Steps	Permite modificar el proceso de compilación sustituyendo los pasos en la compilación normal. Solamente disponible si ha seleccionado un fichero en el panel lateral.	Seleccionado para utilizar custom build steps	Vacío
Exclude From Build	Excluye ficheros en la compilación. Solamente disponible si ha seleccionado un fichero en el panel lateral.	Seleccionado para excluir un archivo en la compilación.	Vacío
Verilog 2001	Utiliza Verilog IEEE 1364-2001 en lugar de IEEE 1364-1995.	Seleccionar para admitir Verilog IEEE 1364-2001.	Vacío
Intermediate files	Subdirectorio donde se almacenan los archivos intermediarios, temporales.	Ruta relativa al directorio desde el directorio del proyecto	Nombre de configuración
Output files	El subdirectorio donde se almacenará la salida (.dll, netlist etc.)	Ruta relativa al directorio del proyecto	Nombre de configuración

7.3.3.1.2 Pestaña: Preprocessor



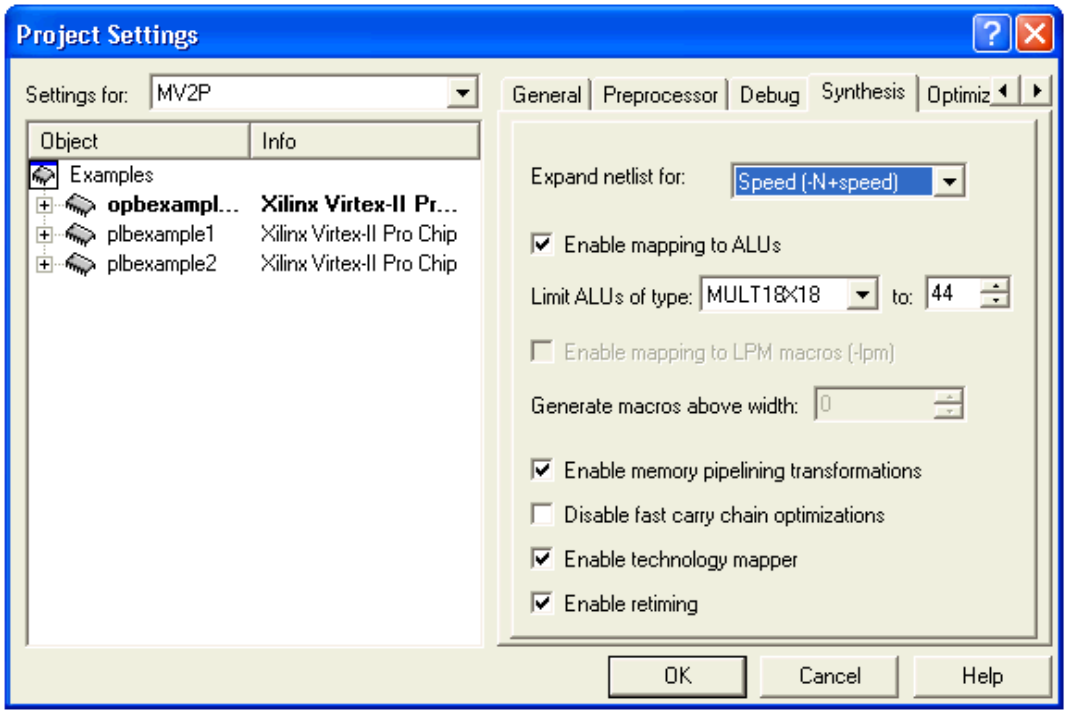
Elemento	Significado	Valor	Defecto
Preprocessor definitions	Equivalente al uso de la directiva #define	Según las necesidades	DEBUG, SIMULATE o NDEBUG
Additional include directories	Añade directorios para incluirlos en el proceso de búsqueda de archivos	Según las necesidades; los directorios van separados por comas	Ninguno
Ignore standard include directories	Permite omitir la búsqueda por defecto de archivos a incluir (para ignorar la inclusión de los archivos estándares).	Seleccionar para omitir la búsqueda por defecto desde la ruta estándar.	Desactivado
Additional preprocessor options	Añadir algún comando de C++	Según las necesidades	Ninguno

7.3.3.1.3 Pestaña: Debug



Elemento	Significado	Valor	Defecto
Working directory	Directorio que el simulador utilizará como directorio de de trabajo.	Ruta relativa al directorio desde el directorio del proyecto.	Directorio actual del proyecto (.)
Detection of simultaneous function calls	Detecta el acceso simultáneo a una función en el proceso de debugaje. Puede utilizar esta opción solamente en el debugaje.	Seleccione para activar.	Activado.
Detection of simultaneous channel reads/writes	Detecta el acceso simultáneo a un canal en el proceso de debugaje. Puede utilizar esta opción solamente en el debugaje.	Seleccione para activar.	Activado.
Detection of simultaneous memory accesses	Detecta el acceso simultáneo a la memoria en el proceso de debugaje. Puede utilizar esta opción solamente en el debugaje.	Seleccione para activar.	Activado.

7.3.3.1.4 Pestaña: Synthesis

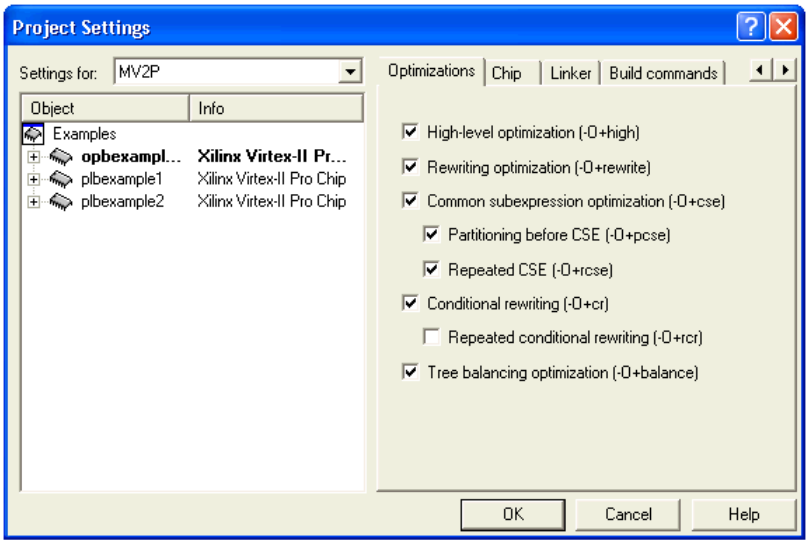


Elemento	Significado	Valor	Defecto
Expand netlist for:	Especifica si el netlist tiene que expandirse para minimizar el área (escoger el área de la lista desplegable), o para maximizar la velocidad (defecto). Esta opción solamente tiene efecto para la salida EDIF para dispositivos Actel.	Selecciona Area vs velocidad.	Speed
Enable mapping to ALUs	Permite al compilador incluir ALUs empotradas (por ejemplo multiplicadores) cuando son disponibles en el dispositivo.	Seleccionar para activar.	Seleccionado para las arquitecturas que tienen ALUs empotradas.
Limit ALUs of type	Limita el número de ALUs empotradas de un tipo específico cuando han sido incluidas en el compilador. Esta opción solamente es permitida si se ha activado el mapeado de la ALU, "mapping to ALUs", en los	Selecciona el tipo de ALU especificando el número máximo de ALUs que se utilizan en el proceso de compilación.	El número máximo de ALUs disponibles en la arquitectura.

	dispositivos con ALUs empotradas.		
Enable mapping to LPM macros (-lpm)	Provoca que el compilador genere macros para operadores comunes (por ejemplo multiplicadores, sumadores) en lugar de expandirlos a nivel de puertas. Las herramientas Place&route pueden utilizar estas macros para optimizar la lógica para dispositivos concretos. La lógica resultante intenta optimizar la velocidad, no obstante puede que incremente el tamaño del diseño.	Seleccionar para activar	Desactivado Esta opción solo esta disponible para EDIF de la familia altera.
Generate macros above Tamaño	Especifica el tamaño/ancho sobre el cual las macros van a ser creadas.	Establece el ancho requerido. Por ejemplo, un valor de 8 significará que las macros se crearán para operadores con más de 8 bits de ancho.	0
Enable memory pipelining transformations	Crea una memoria con acceso segmentado (pipelined memory accesses) para on-chip SSRAM.	Seleccionar para activar	Activado.
Disable fast carry chain optimizations	Desactiva la generación de propagación rápida del acarreo (fast carry chains) para sumadores, restadores, multiplicadores, divisores, comparadores y módulos aritméticos. Fast carry chains tiende a acelerar el diseño, pero crea restricciones en la ubicación de la lógica en un dispositivo.	Seleccionar para desactivar fast carry chains	No activado. Esta opción solo esta disponible para EDIF.
Enable Technology Mapper	Crear salidas EDIF con primitivas que usan tablas de verdad en lugar de utilizar puertas lógicas.	Seleccionar para activar	Seleccionado para arquitecturas de Xilinx y Actel, vacío para el resto. Esta opción solo

			esta disponible para EDIF.
Enable retimer	Intenta reubicar los flip-flops en el circuito intentando satisfacer la restricción marcada por el reloj.	Seleccionar para activar	Seleccionado para arquitecturas Xilinx, vacío para el resto. Esta opción solo esta disponible para EDIF.

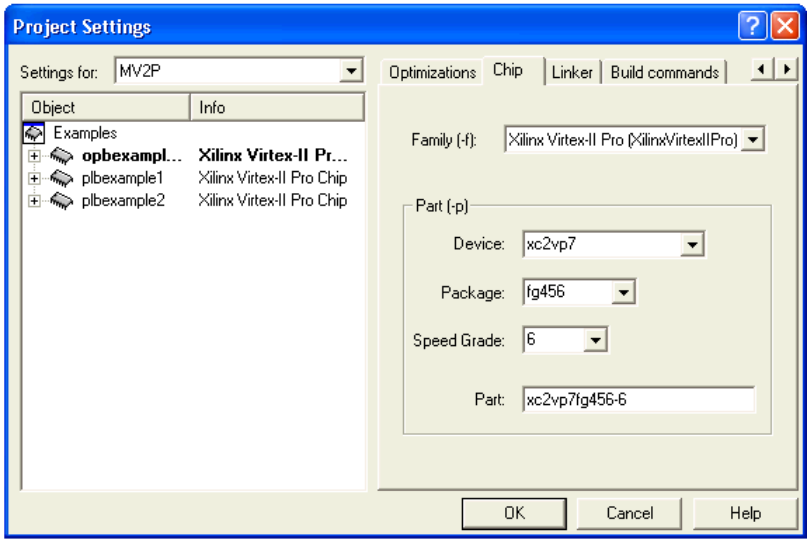
7.3.3.1.5 Pestaña: Optimizations



Elemento	Significado	Valor	Defecto
High-level optimization	Optimización a nivel alto. Acelera la compilación.	Activado si se requiere	No activado en modo Debug. Activado en el resto de modos.
Rewriting optimization	Optimiza la lógica donde las señales son enlazadas de nivel alto a bajo, etc.	Activado si se requiere	Activado. No disponible en modos Debug o Release.
Common sub-Expresión (CSE) optimization	Elimina subexpresiones habituales repetidas. Conduce a diseños más pequeños, no obstante puede que aumente el encaminamiento y por lo	Activado si se requiere	Activado. No disponible en modos Debug o Release.

	tanto retrasa el reloj general.		
Partitioning before CSE optimization	Divide las puertas complejas antes de realizar la optimización CSE.	Activado si se requiere	Activado. No disponible en modos Debug o Release.
Repeated CSE optimization	Repetición de la optimización CSE. Retrasa la compilación.	Activado si se requiere	Activado. No disponible en modos Debug o Release.
Conditional rewriting optimization	Asume ciertos estados y propaga las conclusiones con la lógica. Optimiza según resultados. Retrasará la compilación. Utilizado en conjunción con otras optimizaciones.	Activado si se requiere	Activado. No disponible en modos Debug o Release.
Repeated conditional rewriting optimization	Repite la reescritura condicional hasta que nada más puede ser alcanzado. Puede incrementar substancialmente el tiempo de compilación.	Activado si se requiere	No activado. No disponible en modos Debug o Release.
Tree balancing optimization	Reduce el retraso (delays) en ciertos tipos de código, por ejemplo, arboles sumadores, por medio de la reducción del retraso entre los flip-flops. Desactívelo para dar una relación más clara entre los mensajes de error y el código de fuente.	Activado si se requiere	No activado. No disponible en modos Debug o Release.

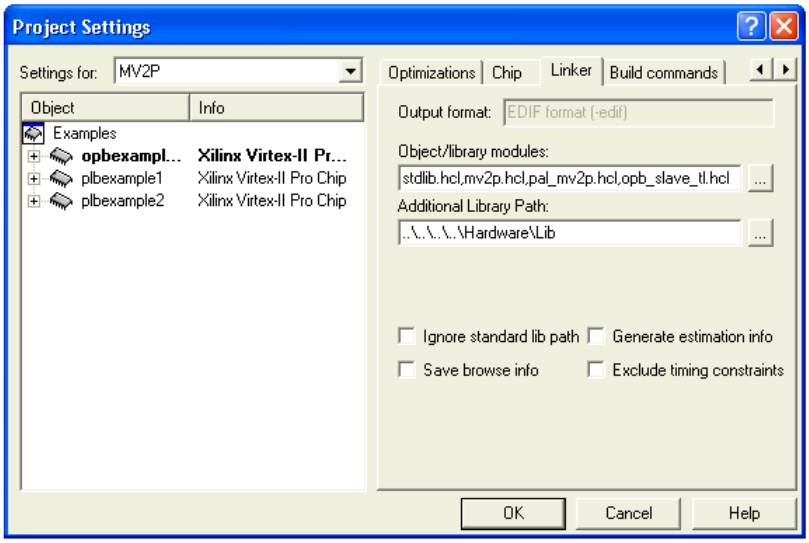
7.3.3.1.6 Pestaña: Chip



Elemento	Significado	Valor	Defecto
Family	La familia que contiene el dispositivo para el que se realiza el diseño.	Seleccionar la familia desde la lista deplegable.	Generic
Device	El dispositivo para el que se realiza el diseño	Seleccionar el dispositivo desde la lista deplegable	Vacío
Package	El empaquetado del dispositvo para el que se realiza el diseño.	Seleccionar el empaquetado desde la lista deplegable	Vacío
Speed Grade	El factor de velocidad del disposito para el que se realiza el diseño.	Seleccionar la velocidad desde la lista deplegable	Vacío
Part	El códigode referencia. Si se introduce manualmente se perderán los ajustes introducidos del dispositivo: grado, velocidad y empaquetado.	Código de referéncia, part number	Depende del proyecto

Se debe especificar la FPGA para obtener el archivo EDIF como salida. En cualquier otro caso, puede seleccionar el valor Generic, para obtener como salida archivos VHDL or Verilog. La salida a un chip generic, no tendrá en cuenta las restricciones impuestas al seleccionar una arquitectura.

7.3.3.1.7 Pestaña: Linker

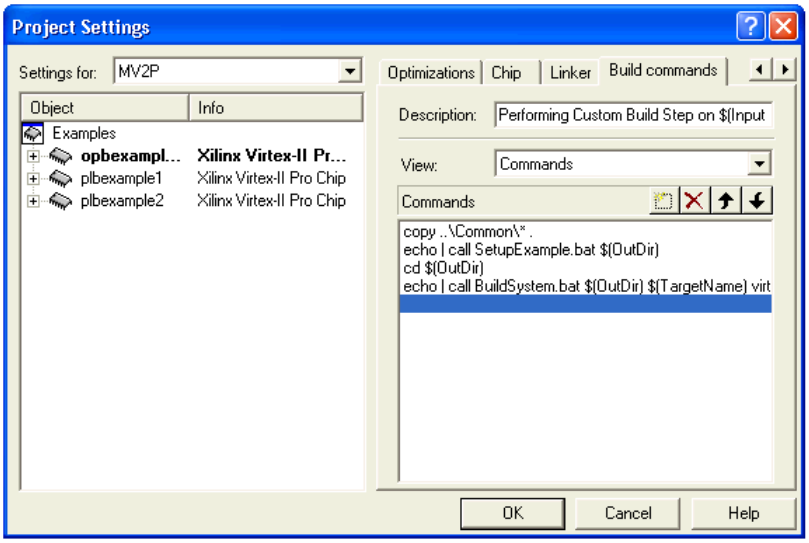


Las opciones que aparecen en esta pestaña dependen del tipo de compilación seleccionada (build configuration).

Elemento	Significado	Valor	Defecto
Output format	Objetivo del compilador	Determinado por las propiedades de target settings	Cuando se necesite.
Object/library modules	Necesidad de incluir librerias extra (.hcl) y archivos objeto extras (.hco)	Escribir la ruta y los nombres de archivos, separados por comas	Cuando se necesite.
Additional Library Path	Ruta de directorio donde se buscarán librerías para Handel-C	Escribir las rutas separadas por comas	Cuando se necesite.
Additional C/C++ Modules	Bibliotecas de C o de C++, y ficheros objeto requeridos por el proyecto.	Escribir la ruta y los nombres de archivos, separados por comas	Ninguno.
VHDL/Verilog output style	Tipo de salida: VHDL o Verilog. (EDIF, VHDL y Verilog no están disponibles en Nexus PDK.)	Active-HDL, Generic, Precision, ModelSim o Synplify. Seleccione ActiveHDL o ModelSim para simulación. Seleccione Generic si desea exportar una herramienta de síntesis que no esté en la lista.	Generis. (solo VHDL y Verilog)

Ignore standard lib path	No buscar las bibliotecas a lo largo de la ruta de las bibliotecas por defecto.	Seleccionar para no buscar en la ruta estándar	Vacío.
Save browse info	Almacenar la información necesaria para explorar símbolos.	Seleccionar para almacenar	Activado.
Generate estimation info	Conseguir que el compilador genere información HTML sobre la profundidad y la información de sincronización (solamente disponible en compilación EDIF)	Seleccionar para activar	Vacío.
Exclude timing constraints	Disable generation of timing constraints (en archivos generados NCF, TCL o ACF)	Seleccionar para desactivar	Vacío (timing constraints are generated)
Simulator compilation command line	Especificar las opciones para el compilador backend. Utilizado para las simulaciones compiladas y, para código en un equipo PC.	Establece como el compilador de C++ es llamado para compilar simulator.dll. Puedes utilizar 4 parámetros proporcionador por el compilador. Puedes también especificar los comandos para generar un archivo de .exe.	Opciones de lincaje definidas en el archivo cl.cf Solamente en Debug, Generic y Release.

7.3.3.1.8 Pestaña: Build commands tab

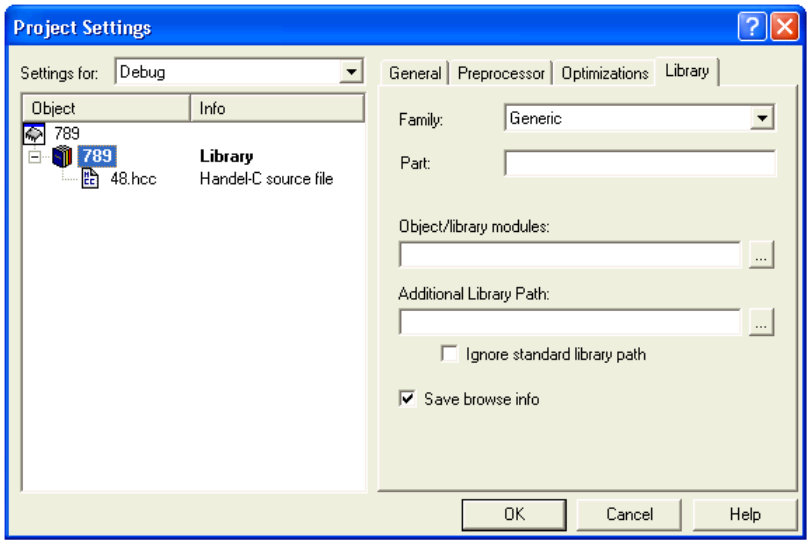


Se pueden añadir comandos específicos para proyectos archivos concretos. Los comandos solamente serán ejecutados allá donde sean especificados.

Los comandos específicos de compilación/generación están disponibles para proyectos, archivos ANSI-C y C++. Si se desea especificar comandos para archivos handel-C, puede especificarlos en las propiedades del proyecto (project settings), en la pestaña “Build commands”.

Description	Especifica la descripción que se mostrará cuando el comando específico es ejecutado en el proceso de construcción. La descripción puede incluir macros de ficheros y directorios.
View	Selecciona Comandos, Salidas o Dependéncias.
Commands/Outputs/ Dependencies	El uso de este panel depende del valor seleccionado en el depleglable “View”.
	Crea un nuevo comando, salida o dependéncia. Presione enter una vez haya finalizado.
	Elimina lo seleccionado: un comando, salida o dependéncia.
	Mover hacia arriba un comando, salida o dependéncia.
	Mover hacia abajo un comando, salida o dependéncia.

7.3.3.1.9 Pestaña: Library



Elemento	Significado	Valor	Defecto
Family	La família que incluye la FPGA.	Seleccionar la familia desde la lista deplegable	La opción genérica es equivalente a omitir la opción -f desde la línea de comandos.
Part	Código de referéncia de la FPGA	Escribir el código de referéncia, part number	Depende del proyecto.
Object/library modules	Requeriemiento de bibliotecas adicionales (.hcl) y ficheros objeto (.hco)	Escribir la ruta y el nombre del fichero separados por comas	Vacío.
Additional library path	Directorios de biblioteca adicionales requeridos	Escribir las rutas separadas por comas	El directorio por defecto es DK\Lib.
Save browse info	Almacenar la información necesaria para explorar símbolos.	Seleccionar para activar	Activado.

7.3.4 Dependencias

Las dependencias se aseguran de que los archivos que no son parte del proyecto sean actualizados durante la compilación. También especifican el orden en que los archivos deben ser compilados y generados.

DK utiliza tres tipos de dependencias:

- Dependencias de proyecto
- Dependencias de archivos
- Dependencias externas

Lo único que se le permite cambiar directamente son las dependencias del proyecto. La otra información la calcula automáticamente el compilador.

7.3.4.1 Dependencias de proyecto

La ventana Project>Dependencies permite cualquier proyecto dentro del entorno de trabajo, siendo reconstruidos cuanto se les requiera.
Si se compila un proyecto complejo como por ejemplo una sistema con múltiples chips, puede realizar un proyecto para cada uno de los chips, y especificar las dependencias entre estos.

7.3.4.2 Dependencias de archivos

Las dependencias de los ficheros se muestran en el dialogo de las propiedades del fichero, ellas especifican:

- El usuario incluye los archivos que no se incluyen en el proyecto pero que se requieren para compilar y construir un archivo seleccionado.
- Otros archivos del proyecto que deben ser compilados antes que este fichero.

Estas dependencias se generan cuando se compila un archivo, se pueden especificar dependencias para un fichero utilizando, procesos de compilación personalizados (custom build steps).

Para examinar las dependencias para un fichero, seleccione el fichero el el entorno de trabajo y posteriormente presiones las teclas Alt+Enter, o bien con el ratón haga click con el botón de propiedades encima del fichero.

7.3.4.3 Dependencias externas

La carpeta externa de las dependencias aparece en la ventana del espacio de trabajo después de que se haya compilado un proyecto. Contiene una lista de los archivos cabecera requeridos por el proyecto que no se incluyen en el proyecto.

7.4 Implementación

Dependiendo del tipo de proyecto que estemos implementando, se puede escoger entre utilizar las librerías proporcionadas por PAL (una plataforma virtual), o bien utilizar una plataforma hardware de celoxica con numerosas funciones ya implementadas para sus tarjetas de desarrollo.

No obstante, se puede implementar un proyecto, si utilizar ninguna de las librerías proporcionadas por Celoxica y construir sus propias funciones.

Sea cual sea el modo de trabajo (Librerias PSL, librerias PAL, sin librerias) siempre estará disponible el debugaje del código en el propio entorno de programación.

En el supuesto de no utilizar las librerías proporcionadas por celoxica, se deberá especificar el pin de la FPGA donde está conectado el dispositivo.

7.4.1 Librería PAL y archivos de cabecera

Para usar PAL se necesita colocar la ubicación de la librería e incluir archivos los archivos para PAL en el entorno de programación DK:

1. Seleccione **Herramientas (Tools) > Opciones (Options)** y abra la etiqueta de **Directorios (Directories).**
2. Presione el botón **Añadir (Add)** y explore hacia:
   InstallDir\PDK\Hardware\Include
   Luego dé un clic sobre **Aceptar (Ok).**

3. Seleccione **módulos de librería (Library modules)** la lista desplegable **directorios de Función para (Show directories for)**.
4. Presione el botón **Añadir (Add)** y explore hacia :
   InstallDir\PDK\Hardware\Lib.
   Luego dé un clic sobre **Aceptar (Ok).**

5. Dé un clic sobre **Aceptar (Ok)** para cerrar el diálogo de la Herramienta Opciones (Options).

Si se quiere trabajar con una plataforma que no está soportada por PAL, se necesitará crear la librería de Soporte de la Plataforma (PSL).

7.4.1.1 Pal Cores

Core	Archivo de cabecera	Librería	Ejemplos
Console	pal_console.hch	pal_console.hcl	console, dumb terminal and keyboard
16-bit Framebuffer	pal_framebuffer16.hch	pal_framebuffer16.hcl	blockstore, framebuffer16 and videoin
8-bit Framebuffer	pal_framebuffer8.hch	pal_framebuffer8.hcl	framebuffer8
Mouse	pal_mouse.hch	pal_mouse.hcl	mouse
Keyboard	pal_keyboard.hch	pal_keyboard.hcl	dumb terminal, keyboard

7.4.1.1.1 Console

Una librería genérica de vídeo para acceder a la consola.

PalConsoleRun

Activar, desactivar o resetear la consola.

PalConsoleEnable

PalConsoleDisable

PalConsoleReset

Limpiar la consola.

PalConsoleClear

Escribir un carácter de texto en la consola.

PalConsolePutChar

Escribir un número hexadecimal sin signo en la consola.

PalConsolePutHex

Escribir un número decimal sin signo en la consola.

PalConsolePutUInt

Escribir una cadena de texto en la consola.

PalConsolePutString

Ejemplo:

```
macro expr ClockRate = PAL_ACTUAL_CLOCK_RATE;

void main (void) {
        /* Create a pointer to a PalConsole structure /*
        PalConsole *ConsolePtr;
        /* Specify the number of video outputs we require */
        PalVideoOutRequire (1);

        par{
                /* Run the Console driver, using PalVideoOutOptimalCT to
                 * select the optimal video output mode for the clock rate. */
                PalConsoleRun (&ConsolePtr, PAL_CONSOLE_FONT_NORMAL,
                        PalVideoOutOptimalCT (ClockRate), ClockRate);
                seq{
                        /* Enable the Console, then print some stuff to it. */
                        PalConsoleEnable (ConsolePtr);
                        UserCode(ConsolePtr);
                }
        }
}

macro proc UserCode(ConsolePtr){
        static ram unsigned char String[32] = "Hello World\n";
        static unsigned Number = 0;

        /* Clear the screen */
        PalConsoleClear (ConsolePtr);

        /* Print a string */
        PalConsolePutString (ConsolePtr, String);
        do {
                PalConsolePutHex (ConsolePtr, Number);
                PalConsolePutChar (ConsolePtr, ' ');
                PalConsolePutUInt (ConsolePtr, Number);
                PalConsolePutChar (ConsolePtr, '\n');
                Number++;
        } while (Number != 20);
}
```

FIG: Ejemplo PAL console

7.4.1.1.2 Framebuffer 8 bits

Librerías gráficas genéricas para frame buffers de 8 bits

PalFrameBuffer8Run

Activar, desactivar o resetear la FrameBuffer 8 bits.

PalFrameBuffer8Enable

PalFrameBuffer8Disable

PalFrameBuffer8Reset

Lectura Escritura de un Pixel

PalFrameBuffer8Read

PalFrameBuffer8Write

Lectura o escritura de cuatro Pixeles adyacentes:

PalFrameBuffer8ReadQuad

PalFrameBuffer8WriteQuad

7.4.1.1.3 Framebuffer 16 bits

Librerías gráficas genéricas para frame buffers de 16 bits.

PalFrameBuffer16Run

Activar, desactivar o resetear la FrameBuffer 16 bits.

PalFrameBuffer16Enable

PalFrameBuffer16Disable

PalFrameBuffer16Reset

Lectura o escritura de un Píxel:

PalFrameBuffer16Read

PalFrameBuffer16Write

Lectura o escritura de dos Pixeles adyacentes:

PalFrameBuffer16ReadPair

PalFrameBuffer16WritePair

7.4.1.1.4 Framebuffer 24 bits

Librerías gráficas genéricas para 24-bit double-buffered frame buffer.

PalFrameBufferDBRun

Activar, desactivar o resetear la FrameBuffer 24 bits.

PalFrameBufferDBEnable

PalFrameBufferDBDisable

PalFrameBufferDBReset

Lectura Escritura de un Pixel

PalFrameBufferDBRead

PalFrameBufferDBWrite

Intercambio de frameBuffers

PalFrameBufferDBSwapBuffers

Establece el valor por defecto del framebuffer

PalFrameBufferDBSetBackground

7.4.1.1.5 Mouse

Una Librería genérica que proporciona el driver para acceder al ratón.

PalMouseRun

Activar, desactivar o resetear el ratón.

PalMouseEnable

PalMouseDisable

PalMouseReset

Establece el intervalo máximo para las coordenadas del ratón:

PalMouseSetMaxX

PalMouseSetMaxY

PalMouseSetMaxZ

Consulta el intervalo máximo para las coordenadas del ratón:

PalMouseGetMaxXTamañoCT

PalMouseGetMaxYTamañoCT

PalMouseGetMaxZTamañoCT

Establece nuevas coordenadas del ratón.

PalMouseSetXY

Activa o desactiva el wrapping.

PalMouseSetWrap

Consulta la posición actual del ratón.

PalMouseGetX

PalMouseGetY

PalMouseGetZ

Consulta el número máximo de botones del ratón.

PalMouseGetMaxButtonCT

Consulta el estado de los botones del ratón.

PalMouseGetButton

Consulta el cambio de posición del ratón.

PalMouseReadChange

Ejemplo:

```
macro expr ClockRate = PAL_ACTUAL_CLOCK_RATE;
void main (void) {
        /* Create a pointer to a PalMouse structure  */
        PalMouse *MousePtr;
        /* Specify the number of PS/2 ports we require */
        PalPS2PortRequire (1);

        par {
                /* Run the mouse driver, using PalPS2PortCT to select
                   the physical PS/2 port to use. */
                PalMouseRun (&MousePtr, PalPS2PortCT (0), ClockRate);
                seq {
                        /* Enable the mouse, then run the user code.  */
                        PalMouseEnable (MousePtr);
                        UserCode (MousePtr);
                }
        }
}
```

```
macro proc UserCode (MousePtr) {
        unsigned MouseX, MouseY;

        /* Setup mouse */
        par {
                PalMouseSetMaxX (MousePtr, 640);
                PalMouseSetMaxY (MousePtr, 480);
                PalMouseSetMaxZ (MousePtr, 31);
                PalMouseSetWrap (MousePtr, 0);
        }
        MouseX = PalMouseGetX (MousePtr);
        MouseY = PalMouseGetY (MousePtr);
        etc ...
}
```

FIG: Ejemplo PAL Mouse

7.4.1.1.6 Keyboard

Una Librería genérica que proporciona el driver para acceder al teclado.
PalKeyboardRun

Activa, Desactiva o resetea el teclado.
PalKeyboardEnable

PalKeyboardDisable

PalKeyboardReset

Leer un carácter ASCII.
PalKeyboardReadASCII

Lee del teclado en modo escaneo
PalKeyboardReadScanCode

Ejemplo:

```
macro expr ClockRate = PAL_ACTUAL_CLOCK_RATE;
void main (void) {
        /* Create a pointer to a PalKeyboard structure */
        PalKeyboard *KeyboardPtr;

        /* Specify the number of PS/2 ports we require (2 because on most
         * Celoxica boards port 0 is mouse, and port 1 is keyboard). */
        PalPS2PortRequire (2);

        par {
                /* Run the keyboard driver, using PalPS2PortCT to select
                 * the physical PS/2 port to use. */
                PalKeyboardRun (&KeyboardPtr, PalPS2PortCT (1), ClockRate);

                seq {
                        /* Enable the keyboard, then run the user code. */
                        PalKeyboardEnable (KeyboardPtr);
                        UserCode (KeyboardPtr);
                }
        }
}

macro proc UserCode (KeyboardPtr) {
        unsigned Char;
        PalKeyboardReadASCII (KeyboardPtr, &Char);
        etc ...
}
```

FIG: Ejemplo PAL Console

7.4.2 PSL 203 y 203e

Para usar PSL se necesita colocar la ubicación de la librería rc203.hcl o rc203e.hcl, e incluir archivos los archivos para PSL en el entorno de programación DK:

1. Seleccione **Herramientas (Tools) > Opciones (Options)** y abra la etiqueta de **Directorios (Directories).**

2. Presione el botón **Añadir (Add)** y explore hacia:

   InstallDir\PDK\Hardware\Include,
   Luego dé un clic sobre **Aceptar (Ok).**

3. Seleccione **módulos de librería (Library modules)** la lista desplegable **directorios de Función para (Show directories for)**.

4. Presione el botón **Añadir (Add)** y explore hacia :

   InstallDir\PDK\Hardware\Lib
   Luego dé un clic sobre **Aceptar (Ok).**

5. Dé un clic sobre **Aceptar (Ok)** para cerrar el diálogo de la Herramienta Opciones (Options).

7.4.2.1 Reloj

Antes de añadir la librería en el código fuente, debe especificar un reloj, hay 4 posibilidades:

- RC203_CLOCK_USER (CLKUSER del generador de reloj),
- RC203_CLOCK_EXPCLK0 (EXPCLK0 del puerto de expansión ATA),
- RC203_CLOCK_EXPCLK1 (EXPCLK1 del puerto de expansión ATA),
- RC203_TARGET_CLOCK_RATE (reloj interno de 2MHz a 300MHz,).

Después de haber establecido la fuente del reloj, y habiendo añadido el archivo rc203.hch o rc203e.hch, se puede especificar el reloj, por ejemplo si desea establecer un reloj de 50MHz:

```
#define RC203_TARGET_CLOCK_RATE = 50000000
#include "rc203.hch"
```

7.4.2.2 Macros disponibles

Las siguientes macros ya están implementadas para realizar el control de los dispositivos de la tarjeta RC203. El nombre de las funciones, así como ¡ los parámetros de entrada/salida están en el archivo “Platform Developer’s Kit, RC200/203 Manual” (RC200_203 Manual.pdf):

- Macros para LEDs
- Macros para pulsar botones

- Macros para displays de 7 segmentos
- Macros para ZBT SRAM
- Macros para puetos PS/2
- Macros para puertos RS232
- Macros para pantallas táctiles
- Macros para salida de vídeo
- Macros para entrada de vídeo
- Macros para entrada/salida de audio
- Macros para tarjetas SMARTMEDIA
- Macros para Ethernet
- Macros para la reconfiguración de la FPGA
- Macros para el control del CPLD
- Macros para la comunicación mediante el puerto paralelo
- Macros para la los pines de expansión

7.5 Simulación

Mediante la compilación del proyecto con el compilador back end, se puede realizar la simulación del código implementado con Handel-C.

Hay dos opciones para realizar la simulación del código:

- Trabajar con el debuger
- Trabajar con PAL

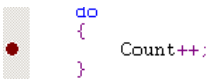
Se puede trabajar siempre con el debugador, para realizar el debugaje de una aplicación consultando diferentes valores del programa, aún cuando se esta ejecutando el simulador PAL:

- Ventada con las variables del programa.
- Ventana con un inspector de los watches añadidos.
- Ventana con los hilos de ejecución paralela
- Ventana con el estado de la Pila

Para lanzar el debuger tan solo se tiene que realizar la compilación del proyecto e iniciar la ejecución.

Para detener el programa en ejecución cuando este se ejecuta en el debugeador, para ir a la instucción anterior.Además se puede ejecutar paso a paso o bien ir a la siguiente intrucción, este proceso se puede realizar de diferentes modos, si se quiere esperar a la salida de la ejecución de una función, si quiere introducirse dentro de una función, si bien desea ejecutar la siguiente instrucción, o bien ejecutar el conjunto de instrucciones hasta llegar donde apunta el cursor:

También existe la posibilidad de insertar “breackpoints”, el programa al encontrarse una línea con un “breackpoint”, detiene su ejecución no abortando la ejecución del programa. Cuando se añade un breackpoint aparece un circulo rojo:



La simulación con la plataforma virtual PAL, permite una rápida implementación de las aplicaciones complejas, sin tener en cuenta la plataforma destino de dicha implementación, para a posteriori reimplementar los algoritmos para una plataforma específica:

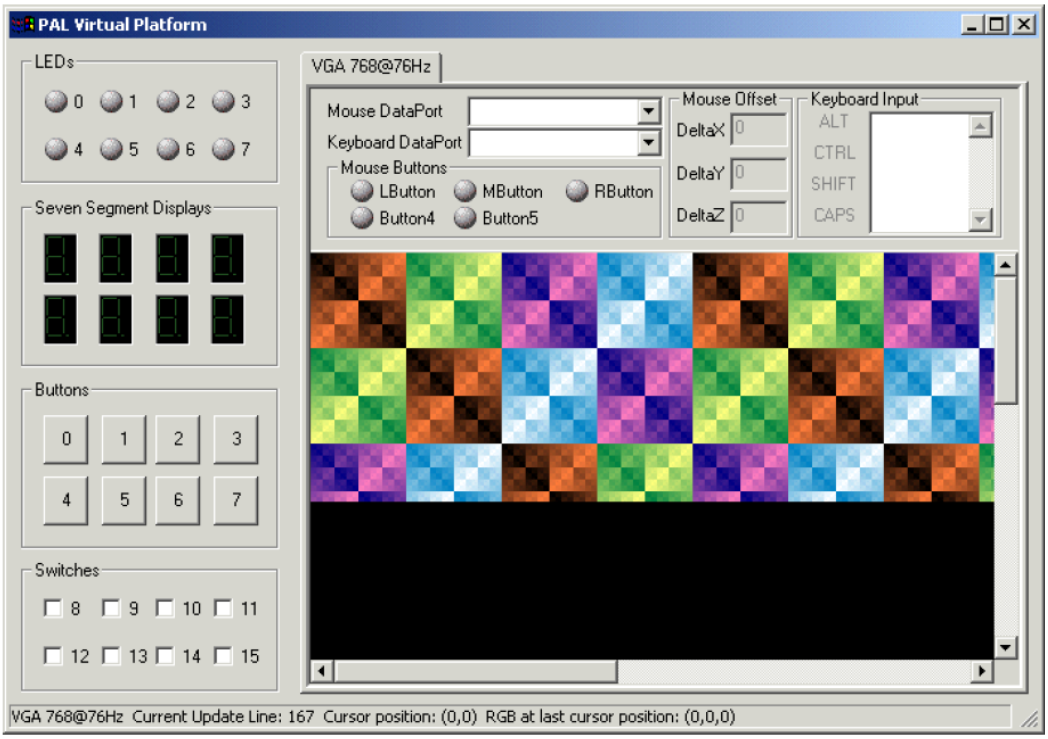


Ilustración 126: Plataforma virtual PAL en ejecución

Con la plataforma virtual se puede realizar la simulación de:

- Hasta 8 LEDs (formulario pantalla).
- Hasta 8 Displays de 7 segmentos (formulario pantalla).
- Hasta 8 Interruptores (formulario pantalla).
- Hasta 8 Botones (formulario pantalla).
- Puertos RS232 (PC).
- Puertos Impresora (PC).
- Ratón (PC).
- Teclado (PC).
- Fast RAM (simulada).
- PL1 Pipelined RAM (simulada).
- Salida de Vídeo. Salidas VGA permitidas (formulario pantalla):
  - 480 a 60Hz de refresco
  - 480 a 75Hz de refresco
  - 600 a 60Hz de refresco
  - 600 a 72Hz de refresco
  - 768 a 60Hz de refresco
  - 768 a 76Hz de refresco
- Entrada de Vídeo. Se pueden simular la siguientes entradas:
  - 160 x 120, 176 x 144, 320 x 240, 352 x 288, 720 x 480 or 720 x 576.
- Audio Entrada/Salida.
- Red, capturando trafico desde la tarjeta del PC.

Para acceder a los ejemplos de PAL, ejecutar el workspace ubicado en: Inicio>Programs>Celoxica>Platform Developer's Kit>PAL>PAL Examples Workspace)

7.6 Síntesis, Place&Route y programación FPGA

La síntesis se realiza a un nivel transparente al programador, dependiendo del modo seleccionado de compilación (EDIF, Simulation, RC203, RC200, ...) y del tipo de salida especificada en las propiedades de compilación, el mismo código puede ser compilado para dar diferentes salidas.

Para realizar la síntesis con el objetivo de obtener un fichero de programación o de reprogramación de una FPGA (bitstream,...) a partir de un fichero sintetizado por la compilación de DK, se aconseja que se utilice las herramientas diseñadas para este propósito proporcionadas por los fabricantes de la FPGA. No obstante DK proporciona la posibilidad de añadir comandos cuando se realiza una cierta compilación de un programa, permitiendo acelerar el proceso de síntesis, P&R y obtención del bitstream.

DK además proporciona posibilidades para optimizar aplicaciones mediante las propiedades de un proyecto. Además permite utilizar varias herramientas de sintetización de diferentes fabricantes, estas herramientas adicionales de sintetización tan solo se requieren cuando se precisa optimizar significativamente el código.

Para ver varios modos de sintetización, compilación, etcétera, diríjase a los ejemplos proporcionados en este manual.

8 Ejemplos

En este capítulo, se ve de un modo más práctico el diseño de una aplicación mediante el uso de la herramienta DK Design Suite de Celoxica y el lenguaje de programación Handel-C. Los ejemplos de este capítulo describen como realizar un contador BCD, siendo los objetivos distintos para cada ejemplo. Los objetivos para cada ejemplo representan las fases de diseño, siendo las metas distintas para cada fase. Para pasar a la siguiente fase (ejemplo) se requiere haber superado la fase (ejemplo) anterior. De este modo se pretende mostrar el uso de las herramientas de Celoxica adecuadas para cada fase.

8.1 Contador BCD con debugaje del código

Mediante el siguiente ejemplo se familiarizará con el uso de las herramientas de desarrollo proporcionadas por DK, implementando un programa para posteriormente realizar su simulación.

8.1.1 Objetivos

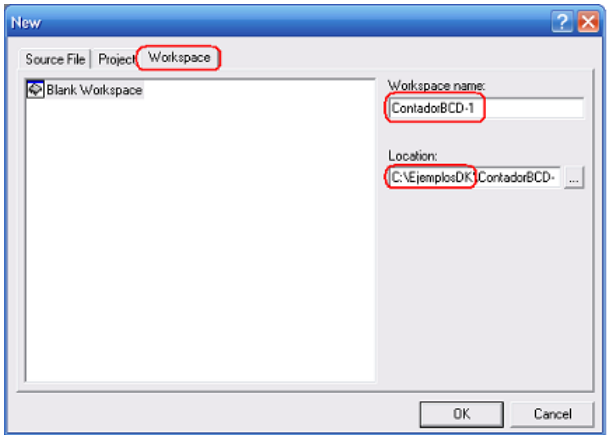
Los objetivos de este ejemplo son:

- Creación de un espacio de trabajo (workspace), un proyecto y un programa escrito en handel-C.
- Compilar un proyecto para simulación.
- Trabajar con el simulador.

8.1.2 Tareas

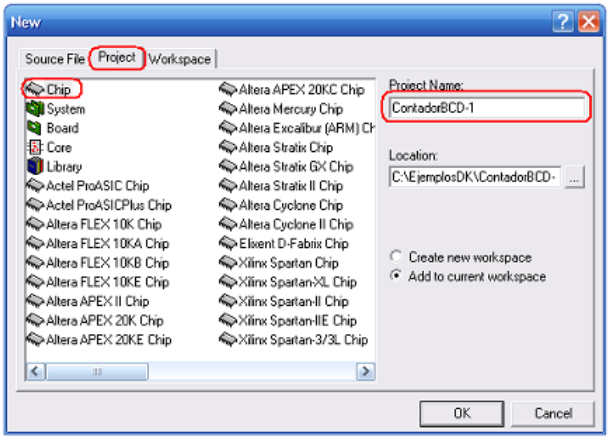
- Abrir DK design Suite de Celoxica:

- Crear un workspace vacío en c:\EjemplosDK llamado “ContadorBCD-1”.



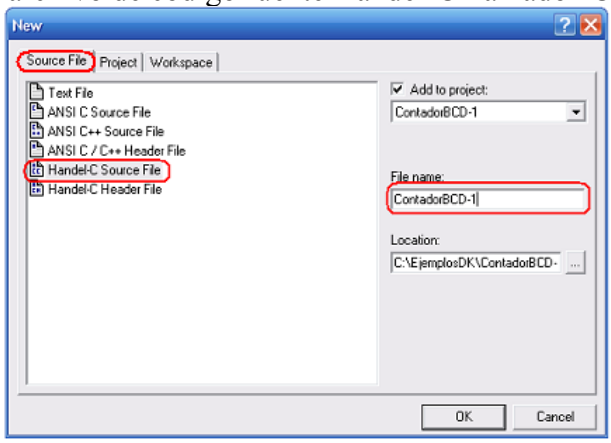
(File->New)

- Crear un nuevo proyecto de tipo “Chip” llamado “ContadorBCD-1”.



(File->New)

- Crear un archivo de código fuente Handel-C llamado “ContadorBCD-1”.



(File->New)

BCD es un modo de representación de los número en binario. Cada uno de los dígitos está representado con 4 bits. Los números del 0 al 9 se muestran a continuación:

Número decimal	Representación BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Se creará un contador de '00' hasta '99' en BCD. Cuando el contador alcance el valor de '99', el contador tendrá que dar la vuelta a '00'. Para que el código compile correctamente se tendrá que incluir la siguiente sentencia al principio del programa:

```
set clock = external;
```

Para almacenar el contador BCD de 2 bits se utilizará una array de 2 elementos, cada uno de ellos con 4 bits:

```
static unsigned 4 Digit[2]={0,0};
```

El código de este ejemplo es el siguiente (archivo ContadorBCD-1.hcc):

```
set clock = external;

void main(){
        static unsigned 4 Digit[2]={0,0};
        while(1){
                if ( Digit[0] == 9 ){
                        if (Digit[1] == 9){
                                Digit[1] = 0;
                                Digit[0] = 0;
                        }
                        else{
                                Digit[1]++;
                                Digit[0]=0;
                        }
                }
                else{
                        delay; // balanceo en el tiempo de procesado
                        delay; // balanceo en el tiempo de procesado
                        Digit[0]++;
                }
        }
}
```

8.1.3 Configuración del simulador

Antes de realizar la configuración debe tener instalado Microsoft VC++ 2005 express edition, y Windows Platform SDK. También ha de estar instalado correctamente en su equipo: Celoxica DK Desdign suite y Celoxica PDK.

Para utilizar Microsoft VC++ 2005 express como compilador Back End, añada las siguientes variables de entorno de sistema:

INCLUDE

"C:\Program Files\Microsoft Visual Studio 8\VC\include"

LIB

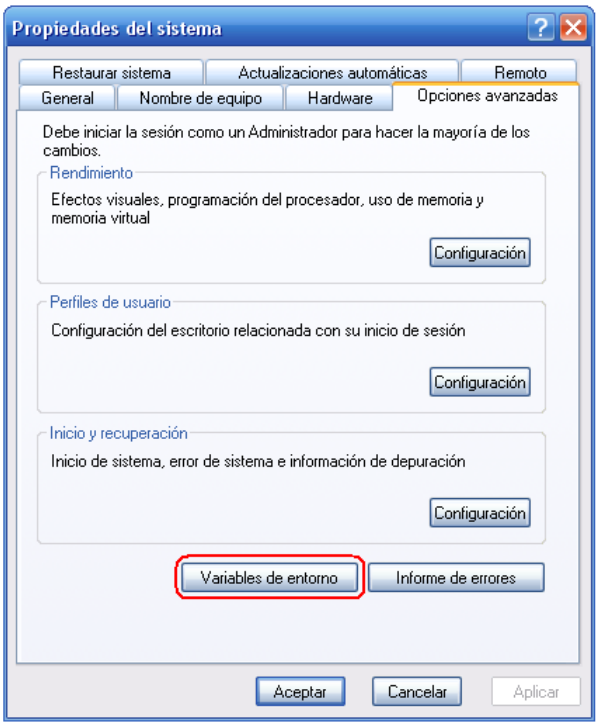
"C:\Program Files\Microsoft Visual Studio 8\VC\LIB"

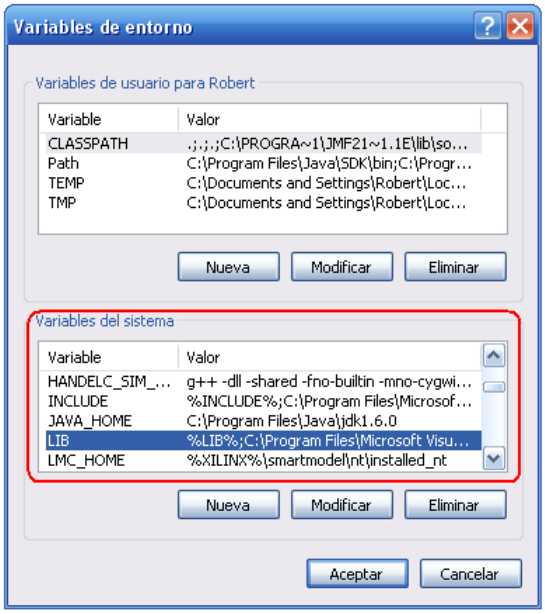
PATH

```
"C:\Program Files\Microsoft Visual Studio 8\VC\bin"
"C:\Program Files\Microsoft Visual Studio 8\Common7\IDE"
```

HANDELC_SIM_COMPILE

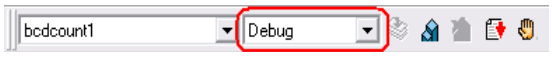
```
cl /Zm1000 /LD /Oityb1 /EHsc /I"C:\ruta\DK\Sim\Include" /Tp"%1" /Fe"%2" %4
```



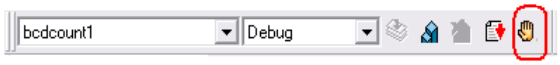


8.1.4 Simulación

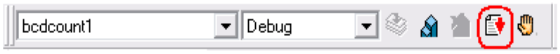
Una vez implementado el código, para realizar la simulación seleccione la configuración Debug,



añada breakpoints al código,



presione F11 para compilar y para iniciar la simulación.



Utilice la ventanas de muestreo de variables, de watch, de call stack y de threads para comprobar que el código funciona correctamente.

Utilice la ventana de control de simulación para detener, continuar o esperar la simulación:

8.1.5 Mejoras en la implementación:

Una primera mejora del código (archivo ContadorBCD-1.hcc), es la reducción de ciclos al incluir rutinas en ejecución paralela. El paralelismo se implementa a nivel de hardware, con lo que realmente se están ejecutando las dos rutinas en paralelo. Ahora tan solo se necesita un ciclo de dalay para compensar las dos ejecuciones dentro de la ejecución condicional.

```
set clock = external;
void main(){
        static unsigned 4 Digit[2]={0,0};
        while(1){
                if ( Digit[0] == 9 ){
                        if (Digit[1] == 9)
                                par{        // se ejecutan paralelamente
                                Digit[1] = 0;
                                Digit[0] = 0;
                                } // fin ejecución paralela
                        else
                                par{        // se ejecutan paralelamente
                                Digit[1]++;
                                Digit[0]=0;
                                } // fin ejecución paralela
                }
                else{
                        delay; // balanceo en el tiempo de procesado
                        Digit[0]++;
                }
```

```
        }
}
```

Una segunda mejora del código (archivo ContadorBCD-1.hcc), es la eliminación de sentencias innecesarias minimizando considerablemente el código:

```
#include "stdlib.hch"
set clock = external;
void main(){
        unsigned (log2ceil (9)) Digit[2];
        while(1){
                if ( Digit[0] == 9 ){
                        Digit[0]=0;
                        if (Digit[1] == 9) Digit[1] = 0;
                        else Digit[1]++;
                }
            else{
                        delay;
                        delay;
                        Digit[0]++;
                }
        }
}
```

Una tercera mejora del código (archivo ContadorBCD-1.hcc), es la inclusión de ejecución paralela para reducir el tiempo de ejecución, (el bloque if que esta dentro de la instrucción par se ejecuta en secuencialmente):

```
#include "stdlib.hch"
set clock = external;
void main(){
        unsigned (log2ceil (9)) Digit[2];
        while(1){
            if ( Digit[0] == 9 ){
                par { // se ejecutan en paralelo
                        Digit[0]=0;
                        if (Digit[1] == 9) Digit[1] = 0;
                        else Digit[1]++;
                } // fin ejecucion paralelo
            }
            else{
                        delay;
                        Digit[0]++;
            }
        }
}
```

8.2 Contador BCD en hardware virtual

Mediante el siguiente ejemplo se familiarizará con el uso de las herramientas de desarrollo proporcionadas por DK, implementando un programa para posteriormente ejecutarlo en hardware virtual.

8.2.1 Objetivos

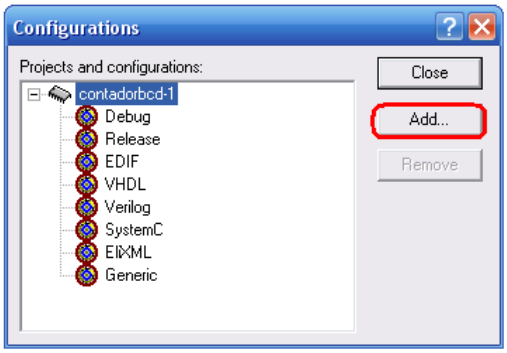
Los objetivos de este ejemplo son:

- Construir un proyecto para ejecutarlo en hardware virtual.

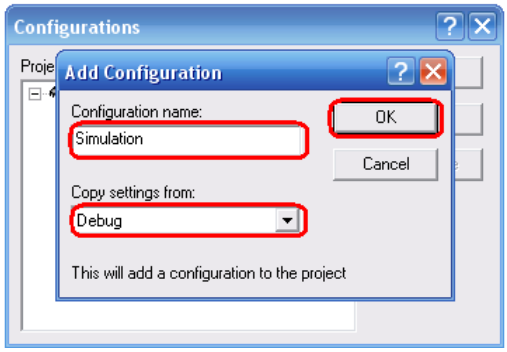
8.2.2 Tareas

- Con el proyecto anterior (ContadorBCD-1).
- Crear una configuración para realizar simulaciones, accediendo mediante la barra de menús: Build → Configurations...

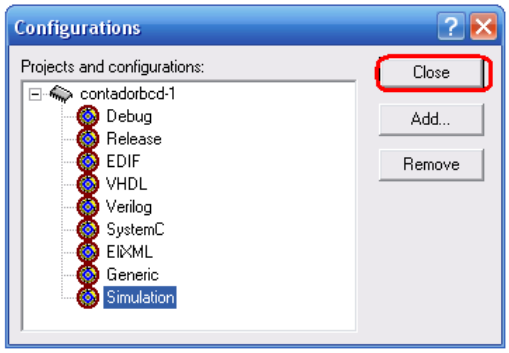
1. Paso: Añadir una nueva configuración



2. Paso: Clonar una configuración des de una ya existente

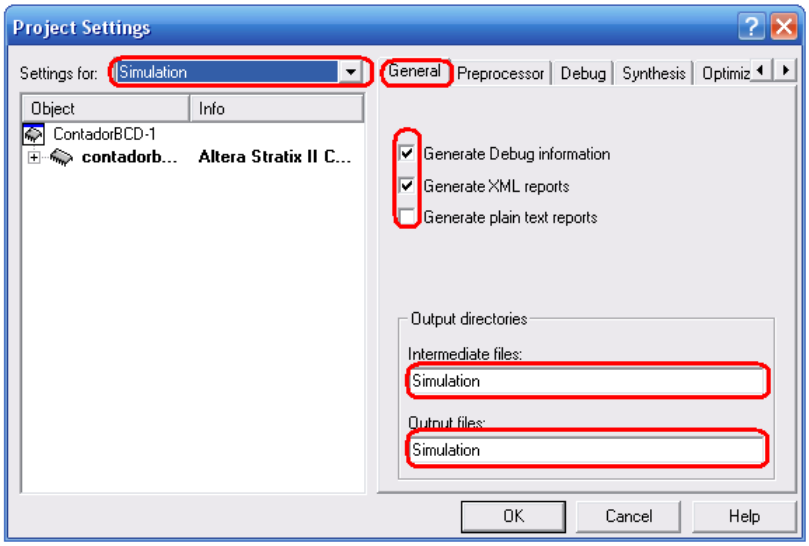


3. Paso: Finalización

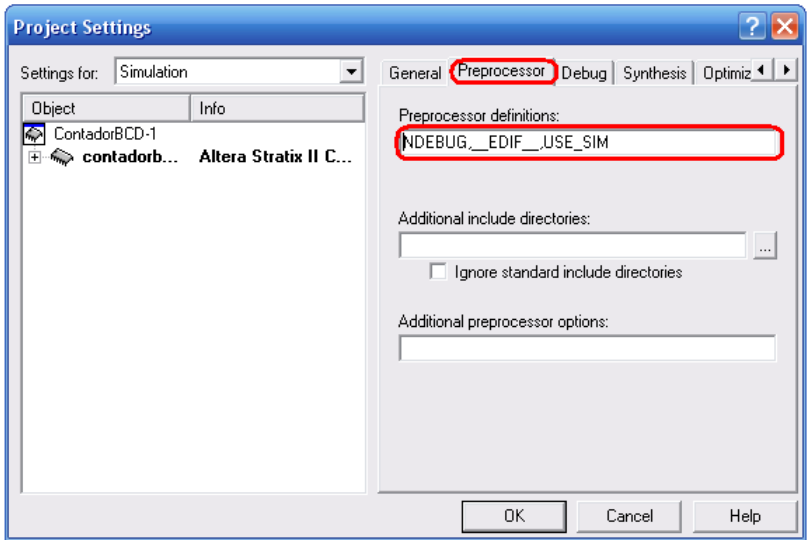


- Configurar la simulación accediendo mediante la barra de menús: Project → Settings...

1. Paso: General

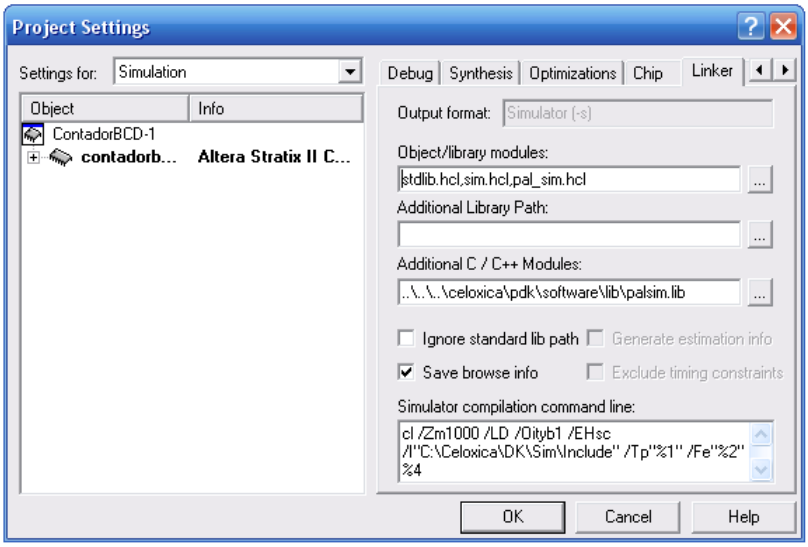


2. Paso: Prepocessor



Preprocessor definitions:
NDEBUG,__EDIF__,USE_SIM

3. Paso: Linker



Object/library modules:
stdlib.hcl,sim.hcl,pal_sim.hcl
Additional C/C++ Modules:
..\..\..\celoxica\pdk\software\lib\palsim.lib
Simulator compilation command line:
cl /Zm1000 /LD /Oityb1 /EHsc /I"C:\Celoxica\DK\Sim\Include" /Tp"%1" /Fe"%2" %4

- Modificación del código:

1. Añadir las librerías para la utilizar la plataforma virtual PAL:

```
#include "pal_master.hch"
```

2. Comprobar la versión de PAL, dentro del programa principal:

```
PalVersionRequire (1, 0);
PalSevenSegRequire (1);
```

3. Modificación del reloj, dentro del programa principal:

```
#define PAL_TARGET_CLOCK_RATE 1000000
```

4. Activar displays de 7 segmentos, dentro del programa principal:

```
PalSevenSegEnable (PalSevenSegCT (2));
PalSevenSegEnable (PalSevenSegCT (3));
```

5. Modificar código para mostrar de los displays, dentro del programa principal:

```
PalSevenSegWriteDigit (PalSevenSegCT (3), Digit[0], 0);
PalSevenSegWriteDigit (PalSevenSegCT (2), Digit[1], 0);
```

6. El código de este ejemplo es el siguiente (archivo ContadorBCD-2.hcc):

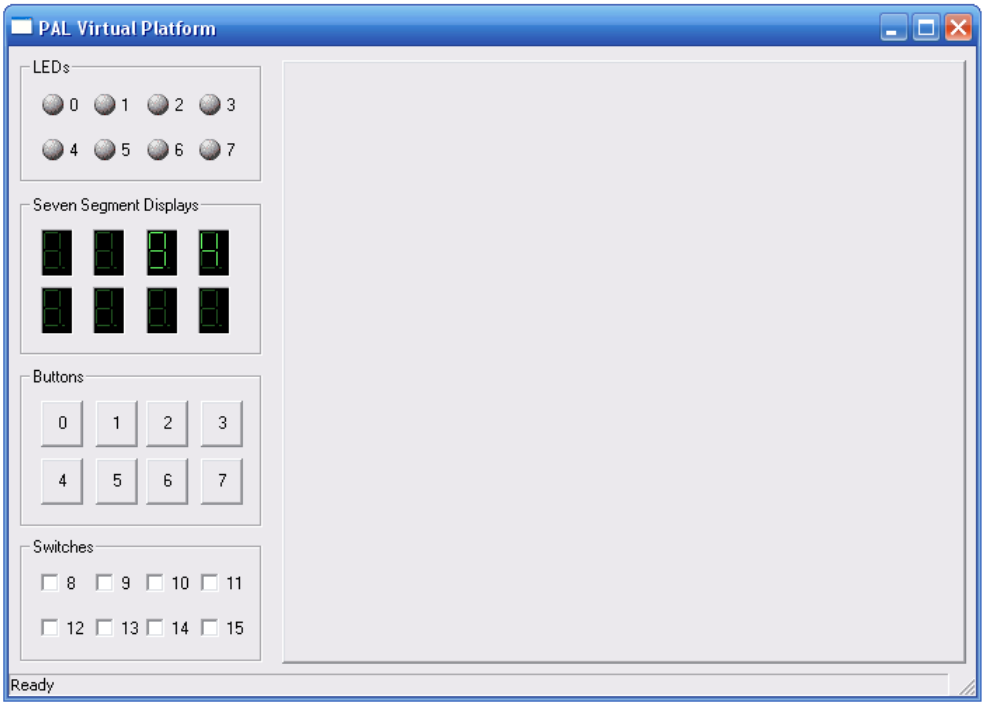
```
#define PAL_TARGET_CLOCK_RATE 1000000
#include "pal_master.hch"
#include "stdlib.hch"
/* Declaración de funciones */
static macro proc Sleep (Milliseconds);
/* Programa principal */
void main (void){
        unsigned (log2ceil (9)) Digit[2];
        /*Comprobar PAL */
        PalVersionRequire (1, 0);
        PalSevenSegRequire (1);
        /* Activar displays */
        PalSevenSegEnable (PalSevenSegCT (2));
        PalSevenSegEnable (PalSevenSegCT (3));
        /* Bucle infinito */
        while (1){
                if ( Digit[0] == 9 ){
                    par { // se ejecutan en paralelo
                            PalSevenSegWriteDigit (PalSevenSegCT (3), Digit[0], 0);
                            PalSevenSegWriteDigit (PalSevenSegCT (2), Digit[1], 0);
                            Digit[0]=0;
                            if (Digit[1] == 9) Digit[1] = 0;
                            else Digit[1]++;
                    } // fin ejecución paralela
                }
                else{
                            // delay;
                    par{// se ejecutan en paralelo
                            PalSevenSegWriteDigit (PalSevenSegCT (3), Digit[0], 0);
                            PalSevenSegWriteDigit (PalSevenSegCT (2), Digit[1], 0);
                            Digit[0]++;
                    }// fin ejecución paralela
                }
                Sleep (99);
        }
}

/* delay de "n" milisegundos */
static macro proc Sleep (Milliseconds){
        macro expr Cycles = (PAL_ACTUAL_CLOCK_RATE * Milliseconds) / 1000;
        unsigned (log2ceil (Cycles)) Count;

        Count = 0;
        do{
                Count++;
        }while (Count != Cycles - 1);
}
```

8.2.3 Simulación

Una vez implementado el código, para realizar la simulación seleccione la configuración simulation,

presione F11 para compilar y para iniciar la simulación.

El resultado de la simulación es el siguiente:



8.3 Contador BCD en Placa RC203

Mediante el siguiente ejemplo se familiarizará con el uso de las herramientas de desarrollo proporcionadas por DK, implementando un programa para posteriormente ejecutarlo en hardware real, una placa de entrenamiento RC203.

8.3.1 Objetivos

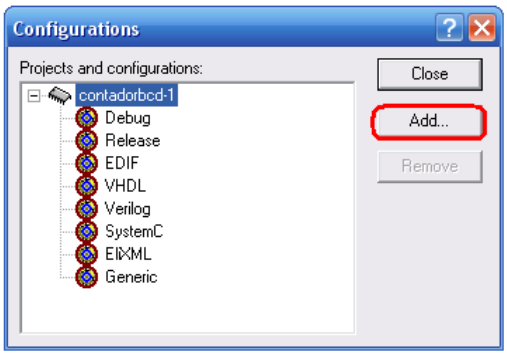
Los objetivos de este ejemplo son:

- Construir un proyecto para ejecutarlo en hardware real.
- Obtener el archivo EDIF.
- Con el archivo EDIF obtener el bitstream, archivo de programación de la FPGA.

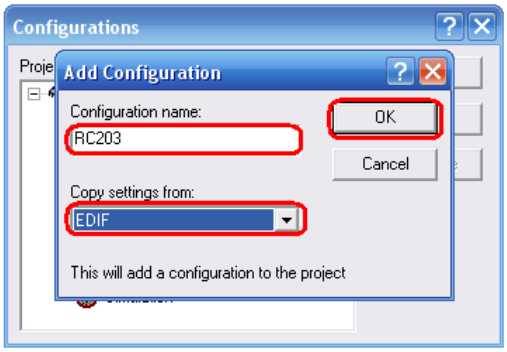
8.3.2 Tareas

- Con el proyecto anterior (ContadorBCD-1).
- Crear una nueva configuración, accediendo mediante la barra de menús: Build → Configurations...

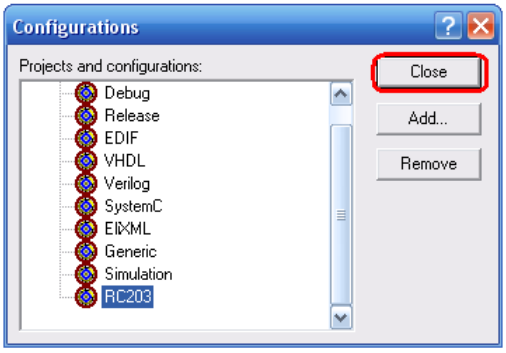
1. Paso: Añadir una nueva configuración



2. Paso: Clonar una configuración des de una ya existente (EDIF)

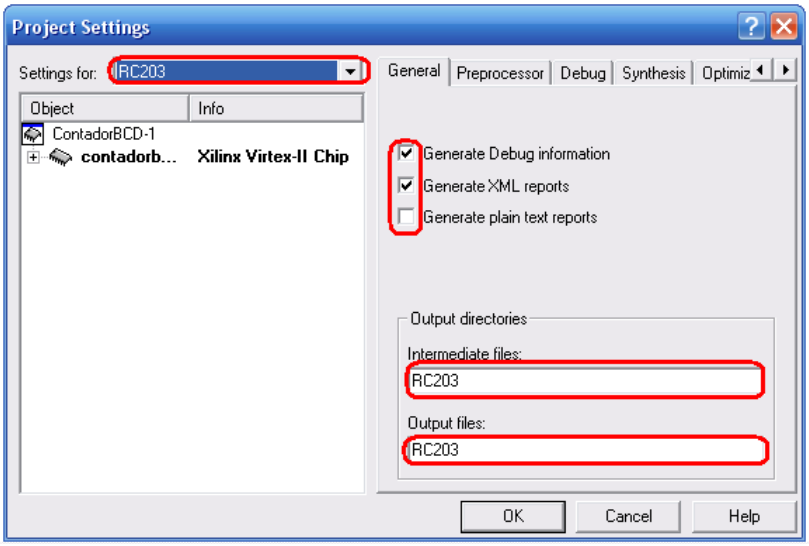


3. Paso: Finalización

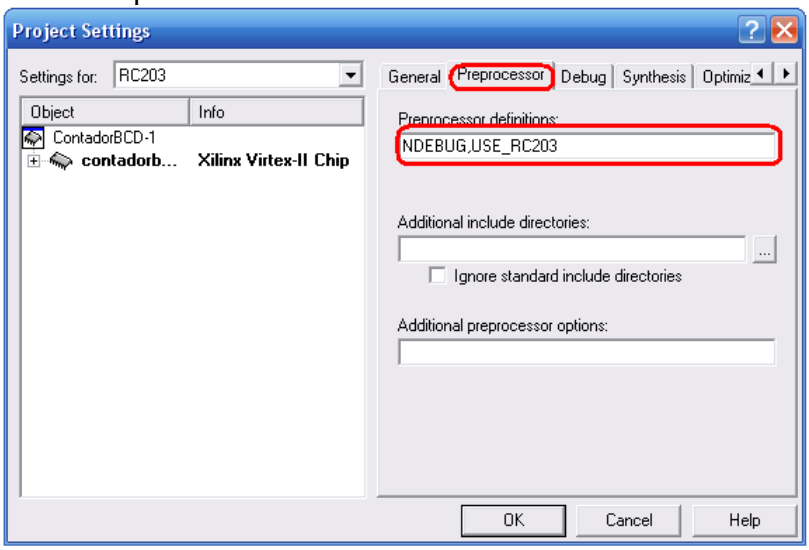


- Configurar la simulación accediendo mediante la barra de menús: Project → Settings...

1. Paso: General

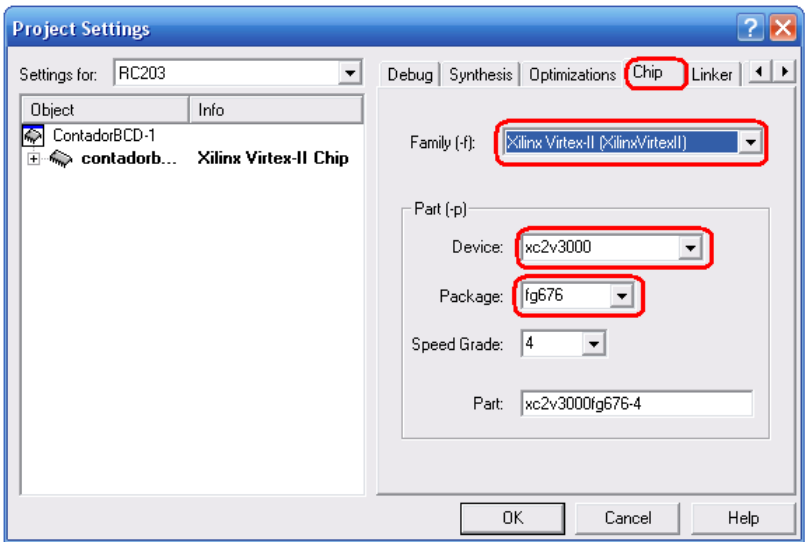


2. Paso: Prepocessor

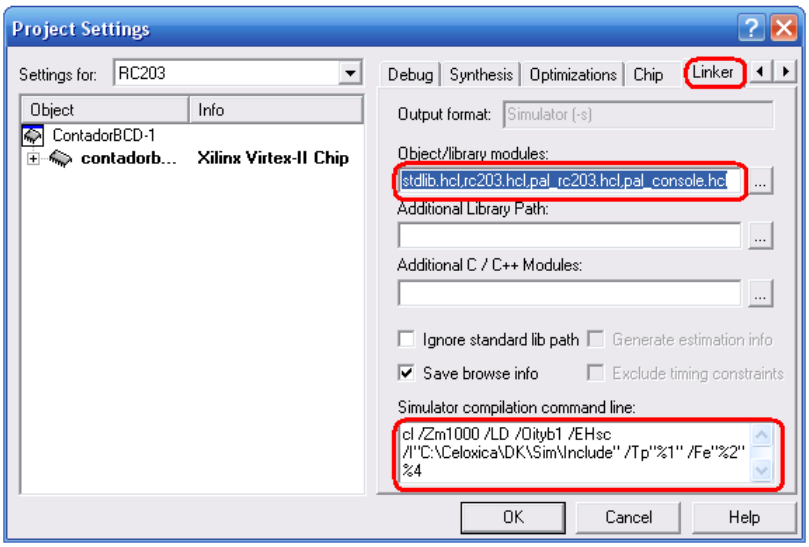


Preprocessor definitions:
NDEBUG,USE_RC203

3. Paso: Chip



4. Paso: Linker



Object/library modules:
stdlib.hcl,rc203.hcl,pal_rc203.hcl,pal_console.hcl
Simulator compilation command line:
cl /Zm1000 /LD /Oityb1 /EHsc /I"C:\Celoxica\DK\Sim\Include" /Tp"%1" /Fe"%2" %4

- Modificación del código:

1. Definición del reloj:

```
#define clockPin     "G14"
#define clockRate    50
set clock = external clockPin with { rate = clockRate };
```

2. Definición del Bus de salida de los diplays de siete segmentos:

```
/* Dos variables para cada uno de los dos displays de 7 segmentos */
 unsigned 7 leftSeg, rightSeg;
```

```
/* Pines donde están conectados los displays a la FPGA */
#define leftSegPins  { "J6", "H5", "M5", "N6", "N5", "K6", "J5" }
#define rightSegPins { "J7", "K5", "N8", "H7", "K7", "L5", "L6" }
```

```
/* Interfaces to link to the pins on the 7-segment displays */
  interface bus_out () leftDisplay  (unsigned 7 out = leftSeg)
            with { data = leftSegPins };
  interface bus_out () rightDisplay (unsigned 7 out = rightSeg)
            with { data = rightSegPins };
```

3. Tabla de conversión de BCD a display de 7 segmentos:

```
/* Two ROMs used to convert between 4-bit BCD and 7-bit integers*/
static rom unsigned 7 numbers0[10] =
            /* 0     1     2     3     4     5     6     7     8     9 */
            { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
static rom unsigned 7 numbers1[10] =
            /* 0     1     2     3     4     5     6     7     8     9 */
            { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
```

4. Código resultante:

```
#include "stdlib.hch"
#define clockPin     "G14"
#define clockRate    50
set clock = external clockPin with { rate = clockRate };
#define leftSegPins  { "J6", "H5", "M5", "N6", "N5", "K6", "J5" }
#define rightSegPins { "J7", "K5", "N8", "H7", "K7", "L5", "L6" }

/* Main program */
void main (void){
        unsigned (log2ceil (9)) Digit[2];
        /*Dos variables para cada uno de los dos displays de 7 segmentos */
        unsigned 7 leftSeg, rightSeg;
        /*Interfaces to link to the pins on the 7-segment displays */
        interface bus_out () leftDisplay  (unsigned 7 out = leftSeg)
                        with { data = leftSegPins };
        interface bus_out () rightDisplay (unsigned 7 out = rightSeg)
                        with { data = rightSegPins };
        /*Two ROMs used to convert between 4-bit BCD and 7-bit integers */
        static rom unsigned 7 numbers0[10] =
            /* 0     1     2     3     4     5     6     7     8     9 */
            { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
        static rom unsigned 7 numbers1[10] =
            /* 0     1     2     3     4     5     6     7     8     9 */
            { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
 /*Bucle infinito */
 while (1){
                if ( Digit[0] == 9 ){
                        par { // se ejecutan en paralelo
                                rightSeg = numbers0[Digit[0]];
                                leftSeg  = numbers1[Digit[1]];
                                Digit[0]=0;
                                if (Digit[1] == 9) Digit[1] = 0;
                                else Digit[1]++;
                        } // fin ejecucion paralelo
                }
```

```
                    else{
                        // delay;
                        par{ // se ejecutan en paralelo
                            rightSeg = numbers0[Digit[0]];
                            leftSeg  = numbers1[Digit[1]];
                            Digit[0]++;
                        } // fin ejecucion paralelo
                    }
                }
            }
```

8.3.3 Obtención del archivo EDIF

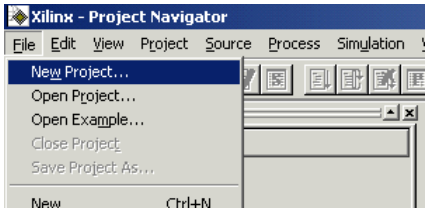
Una vez implementado el código, para realizar la compilación seleccione la configuración RC203,

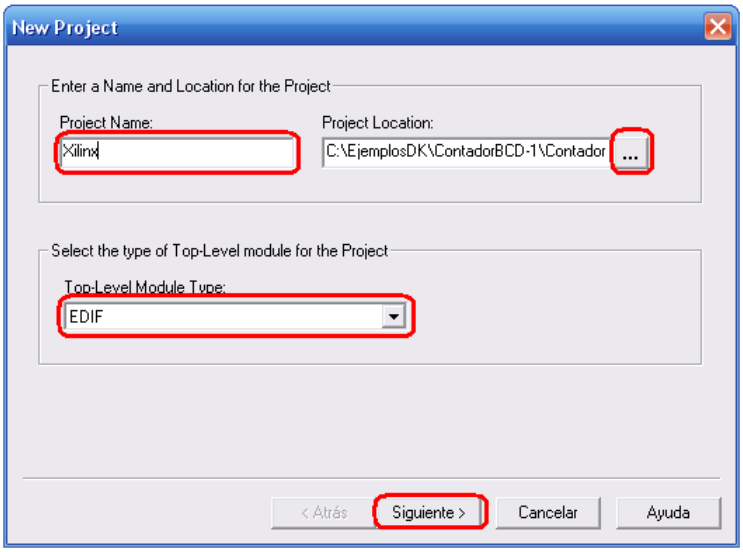
compilar el código.

El resultado de la compilación es el archivo EDIF: contadorbcd-1.edf.

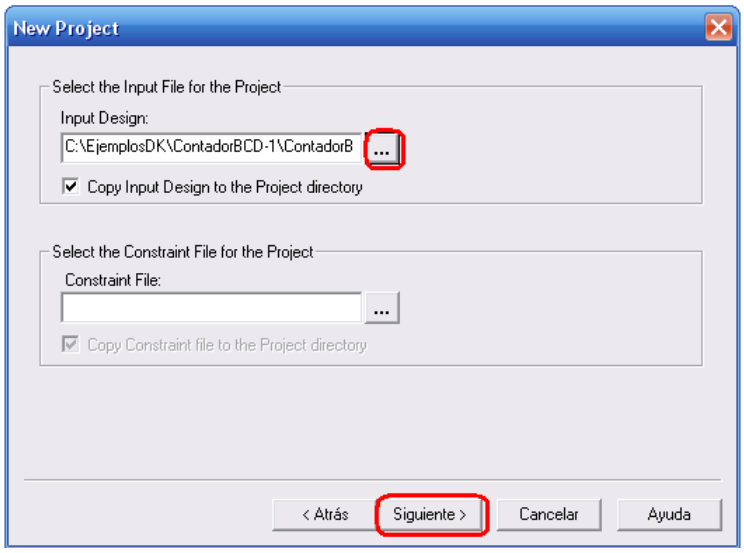
8.3.4 Obtención del archivo bitstream

1. PASO: Crear un nuevo proyecto con Xilinx ISE 7.1i:

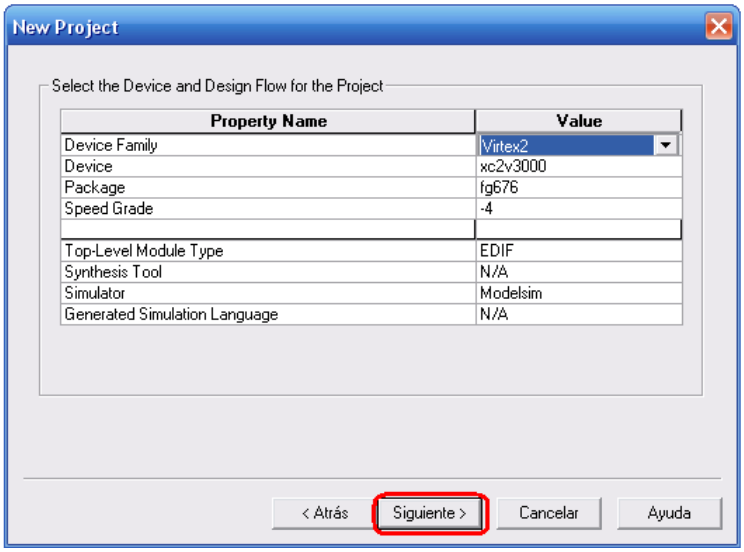




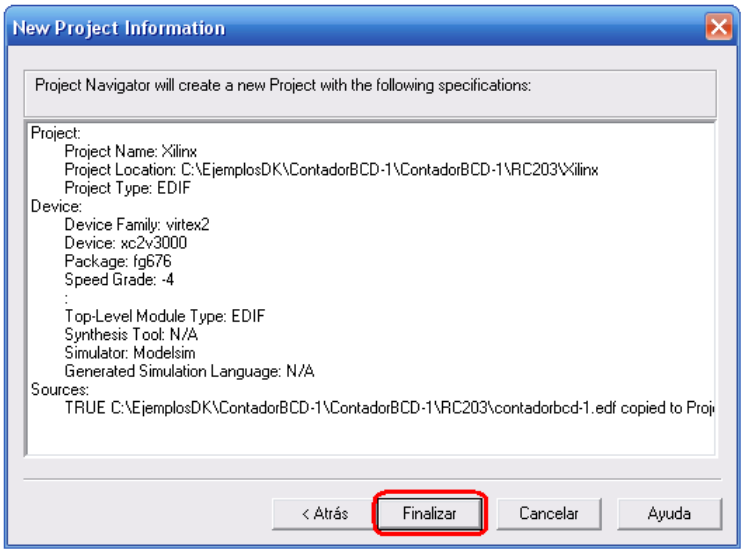
2. PASO: Especificar la ruta al archivo EDIF



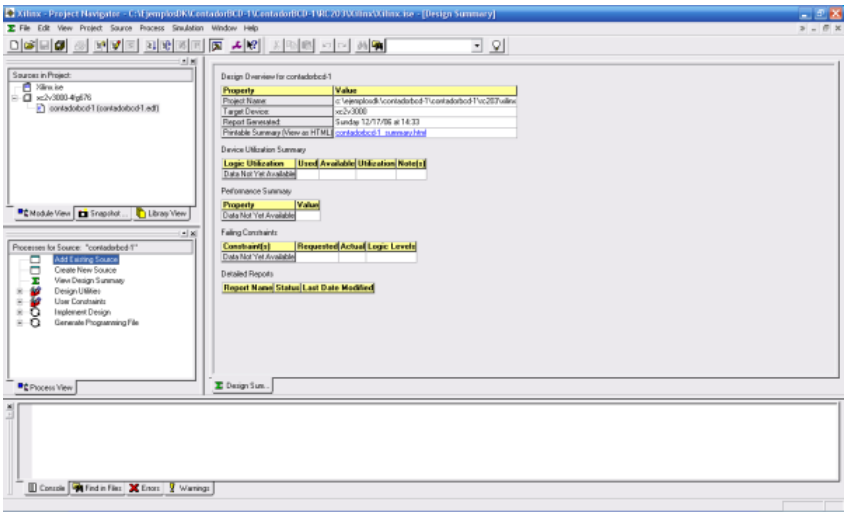
3. PASO: Comprobación de la FPGA



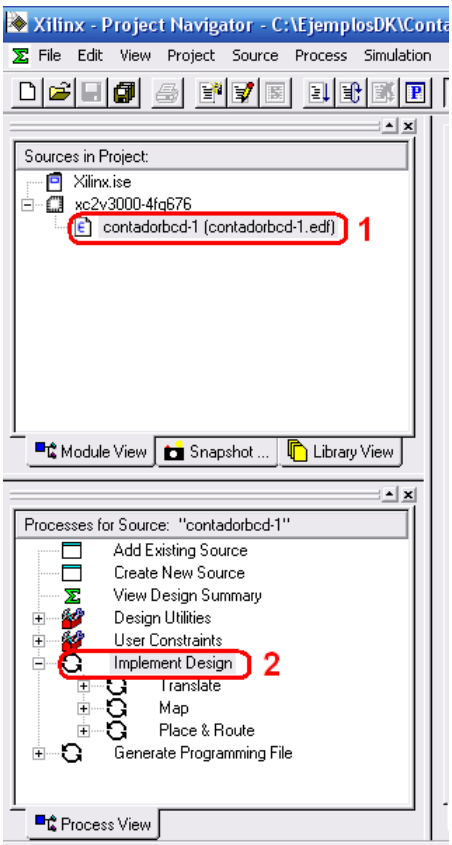
4. PASO: Pantalla resumen del proyecto



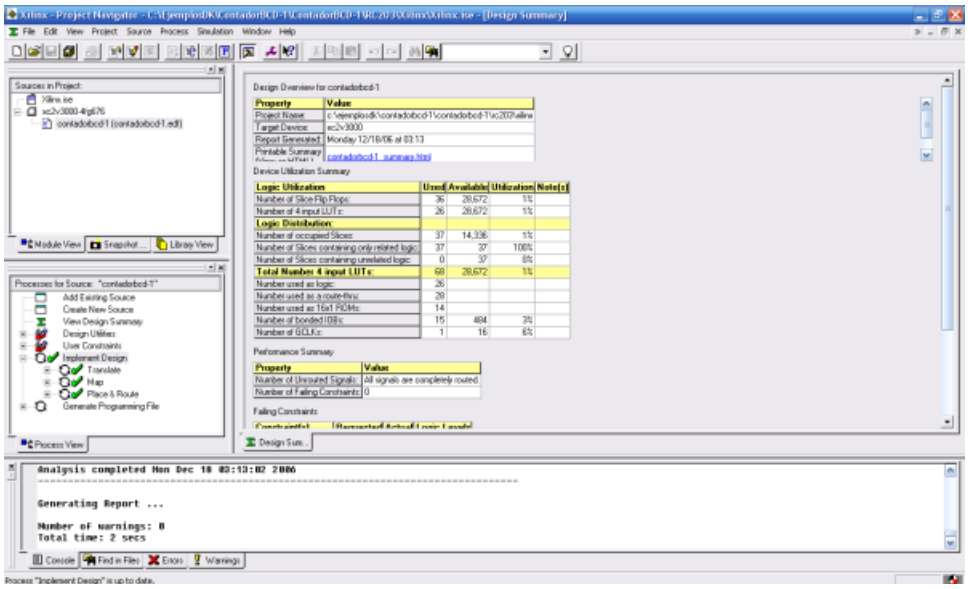
5. PASO: Automáticamente se abrirá esta pantalla:



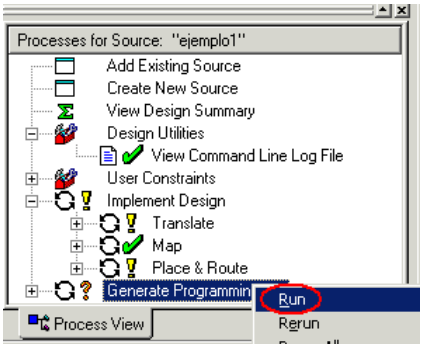
6. PASO: Place & Route
   Seleccionar 1, después seleccionar 2, con el botón de propedades del ratón encima de 2, ejecutaremos “Run”



7. PASO: resultado de P&R

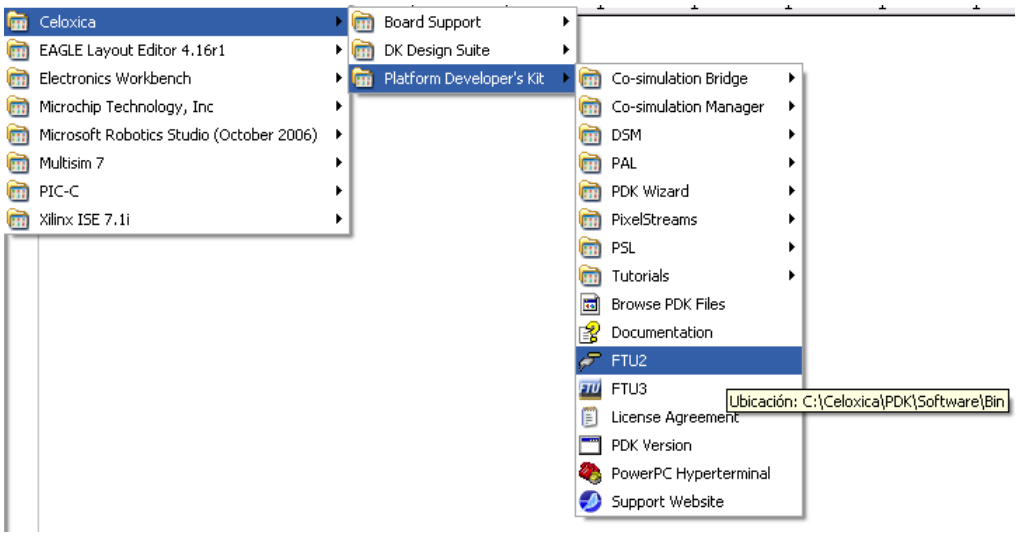


8. PASO: Obtener archivo bitstream, con el botón de propiedades del ratón encima de “Programming file..” ejecutaremos “Run”.

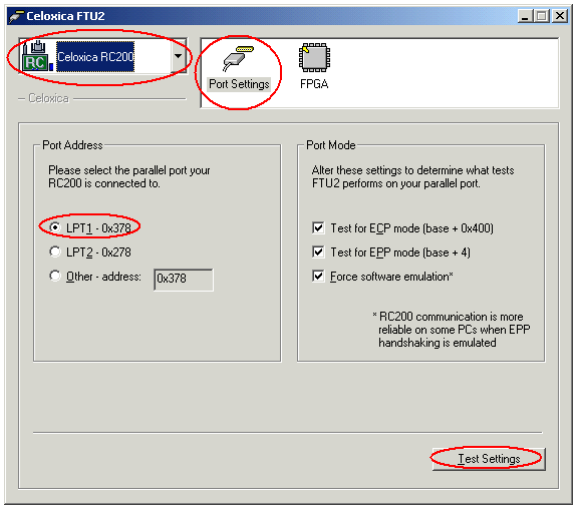


8.3.5 Cargar el archivo en la FPGA

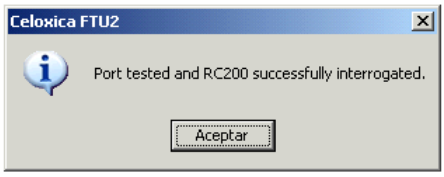
Ejecutar la aplicación FPU2:



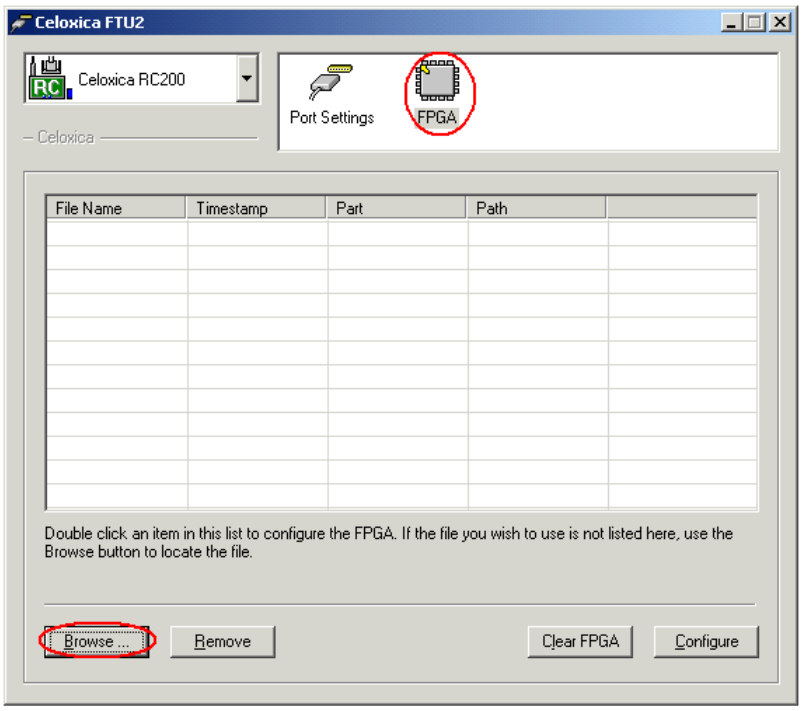
Seleccionar la FPGA, así como el puerto de comunicación LPT:



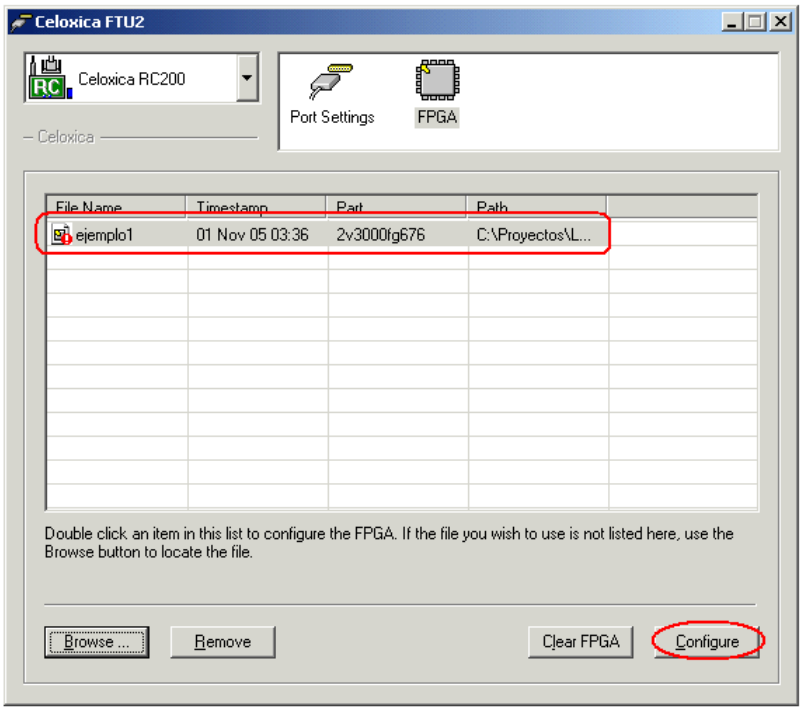
Una vez realizado el testeo de la tarjeta (comunicación), el resultado tiene que ser:



El paso siguiente sera seleccionar el archivo para cargar en la FPGA,



Una vez seleccionado, aparecerá en la lista. “Configure” para cargarlo a la FPGA.



8.3.6 Mejoras en la implementación:

Una mejora del código (archivo ContadorBCD-1.hcc), es la rutina “macro proc BCDCount(digits)” en en ejecución paralela con “rightSeg = numbers0[digits[0]];“ y “leftSeg = numbers1[digits[1]];”. Otra mejora es utilizar el retraso entre el aumento del contador por medio de la variable contador “pauseCount”, su valor estará entre 0 y 0x3FFFFF, y una vez alcanzado su valor máximo, automáticamente volverá a valer 0, y así sucesivamente.

```
#include "stdlib.hch"
#define clockPin     "G14"
#define clockRate    50
set clock = external clockPin with { rate = clockRate };
#define leftSegPins  { "J6", "H5", "M5", "N6", "N5", "K6", "J5" }
#define rightSegPins { "J7", "K5", "N8", "H7", "K7", "L5", "L6" }
/*Declaración de funciones*/
macro proc BCDCount(digits);

/* Programa principal*/
void main (void){
        unsigned (log2ceil (9)) Digit[2];
        /* Two variables for each of the 7-segment displays */
        unsigned 7 leftSeg, rightSeg;
        /*Interfaces to link to the pins on the 7-segment displays */
        interface bus_out () leftDisplay  (unsigned 7 out = leftSeg)
                with { data = leftSegPins };
        interface bus_out () rightDisplay (unsigned 7 out = rightSeg)
                with { data = rightSegPins };
        /*Two ROMs used to convert between 4-bit BCD and 7-bit integers */
        static rom unsigned 7 numbers0[10] =
                /* 0       1       2       3       4       5       6       7       8       9 */
                { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
                static rom unsigned 7 numbers1[10] =
                /* 0       1       2       3       4       5       6       7       8       9 */
                { 0x3f, 0x06, 0x5b, 0x4f, 0x66, 0x6d, 0x7d, 0x07, 0x7f, 0x6f };
        /*At 50MHz, there are 50,000,000 cycles every second
         * We will pause for 2^26 = 67,108,864 cycles, approx 1.3 seconds
         * At 80Mhz, there are 80,000,000 cycles every second
         * We will pause for 2^26 = 67,108,864 cycles, approx 0.8 seconds */
           static unsigned 26 pauseCount = 1;

        /*Bucle infinito*/
        while (1){
                par{    rightSeg = numbers0[Digit[0]];
                        leftSeg  = numbers1[Digit[1]];

                        if(pauseCount == 0)
                                BCDCount(Digit);
                        else    /* Prevent a potential combinational cycle by adding
                                 * a delay for when the if condition is false */
                                delay;

                        pauseCount++;
                }
        }
}

macro proc BCDCount(Digit){
        if (Digit[0] == 9){
                /*Use par statement to make the function run in one clock cycle*/
                par{
                        Digit[0] = 0;
                        if(Digit[1] == 9)   Digit[1] = 0;
                        else   Digit[1]++;
                }
        }
        else
                Digit[0]++;
}
```

Bibliografía

FPGA: Nociones básicas e implementación
M. L. López Vallejo y J. L. Ayala Rodrigo
Departamento de ingeniería y electrónica
Universidad Politécnica de Madrid

Hardware Dinámicamente Reconfigurable
Julio Septién del Castillo y Hortesia Mecha López
Departamento de Arquitectura de Computadores y Automática
Universidad Computense de Madrid

Virtex 2.5V: Field Programmable Gate Arrays
DS003-2 (v2.9.1) December 9, 2002
http://www.xilinx.com

Virtex-II Platform FPGA User Guide
UG002 (v2.0) March 23, 2005
http://www.xilinx.com

Virtex-II Platform FPGAs: Complete Data Sheet
DS031 (v3.4) March 1, 2005
http://www.xilinx.com

VHDL - Lenguaje para descripción y modelado de circuitos
Ingeniería Informática, Universitat de Valéncia
Fernando Pardo Carpio, 14 de octubre de 1997

Diseño de Sistemas Digitales con VHDL
http://www.dte.uvigo.es/vhdl/home.html
S.A. Pérez, E. Soto, S. Fernández
Universidad de Vigo

Diseño Digital, Tema 1. Lenguaje VHDL
Departamento de Sistemas Electrónicos y de Control
Universidad Politécnica de Madrid

Handel-C Language Reference Manual
Copyright © 2004 Celoxica Limited. All rights reserved.
Authors: SB
Document number: RM-1003-4.2

Handel-C Language Reference Manual
Copyright © 2005 Celoxica Limited. All rights reserved.
Authors: RG
Document number: RM-1003-4.2

Using Handel-C with DK
Ashley Sutcliffe, Version 1.0.2
Copyright © 2004 Celoxica Limited. All rights reserved.

Using Handel-C with DK, Advanced Course
Ashley Sutcliffe, Version 1.0.2
Copyright © 2004 Celoxica Limited. All rights reserved.

Uso de Handel-C con DK
Ashley Sutcliffe, Version 1.0.1
Copyright © 2004 Celoxica Limited. All rights reserved.
Tradución al español: José Ignacio Martínez Torre
Informática, Estadística y Telemática (DIET)
Universidad Rey Juan Carlos (URJC), Madrid – ESPAÑA

Uso de Handel-C con DK, Curso Avanzado
Ashley Sutcliffe, Version 1.0.1
Tradución al español:
José Ignacio Martínez Torre
Informática, Estadística y Telemática (DIET)
Universidad Rey Juan Carlos (URJC), Madrid – ESPAÑA

Platform Developer's Kit, PAL Cores manual
Authors: RG
Copyright © 2005 Celoxica Limited. All rights reserved.

Platform Developer's Kit, RC200/2003 manual
Authors: SB
Copyright © 2004 Celoxica Limited. All rights reserved.

Tutorial Xilinx MicroBlaze-uCLinux
Aguayo E, González I. y Boemo E.
Escuela Politécnica Superior, Universidad Autónoma de Madrid, España
JCRA 2004

Celoxica Technology Focus
http://www.celoxica.com/technology/default.asp

DIGITAL FUNDAMENTALS
THOMAS L. FLOYD
Pretince Hall internacional Editions, 1994
ISBN 0-13-228677-7

Fundamentals of Digital Logic with VHDL Design
Stephen Brown, Zvonko Vranesic
Ed. Mc Graw Hill

Digital Signal Pocessing with Field Programmable Gate Arrays
U. Meyer-Baese
Springer-Verlag Berlin Heidelberg New York,
ISBN 3-540-41341-3

ASIC AND FPGA verification: A guide to component modeling
Richard Munden
Morgan Kaufmann, 2005
ISBN 0-12-510581-9

Diccionario de la lengua española
Vigésima segunda edición
© Real Academia Española, 2003

Fundación Wikimedia (***Wikimedia Foundation, Inc.***)
http://es.wikipedia.org/

Green Socs - Open Source SystemC Infrastructure
http://www.greensocs.com/SystemC